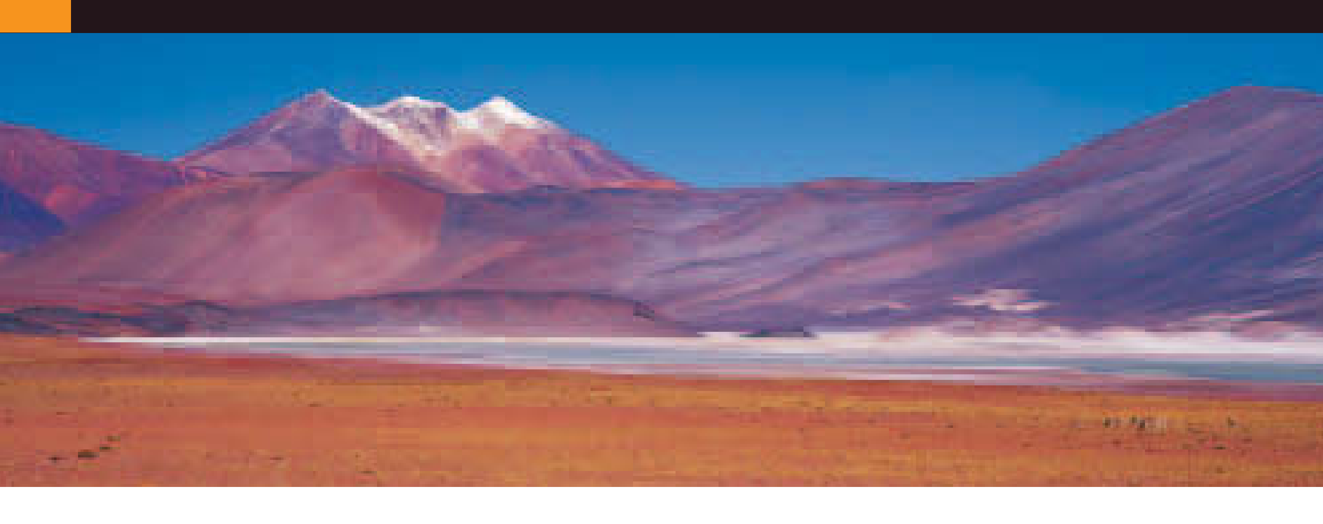
ÉTUDES & DÉVELOPPEMENT

EJB 3

Des concepts à l'écriture du code
Guide du développeur



Laboratoire SUPINFO
des technologies Sun



Préface d'Alexis Moussine-Pouchkine

DUNOD

EJB 3

Des concepts à l’écriture du code
Guide du développeur

EJB 3

Des concepts à l'écriture du code
Guide du développeur

Laboratoire SUPINFO
des technologies Sun

Cet ouvrage a été rédigé par
Frédéric Chuong, Olivier Corgeron
Cyril Joui, Jean-Baptiste Renaux et Maxime Vialette

Préface de
Alexis Moussine-Pouchkine
Architecte Java
de Sun Microsystems France



Illustration de couverture : digitalvision®



ISBN 2 10 050623 4

Préface

Je dois dire que je n'ai pas l'habitude de rédiger de la prose qui commence par « Je » (de la poésie non plus me direz-vous...), mais il semble que ce soit ce qu'on attend d'une préface. Voilà donc mon « je ».

Dans tout le brouhaha qui peut accompagner la sortie d'une technologie informatique, il peut être difficile de distinguer la partie utile du signal. Chacun procède alors avec son expérience pour se faire une idée. Un collègue me disait récemment qu'il fallait toujours attendre la troisième version d'une technologie ou d'un produit avant qu'il ou elle ne soit réellement mûr (en s'appuyant sur de nombreux exemples dans l'informatique de ces dix dernières années). Difficile de dire pour Java quand ce tournant a eu lieu, « Java 2 » date de 1999 et le saut de 1.4 à 5.0 remonte à 2004...

En revanche pour les EJB tout est clair. La version 3.0 est la bonne. Ce qui rend cette version si digne d'intérêt est très certainement la réunion de l'approche POJO, de la configuration par exception et de la nouvelle API de persistance, simple et élégante dont la portée va bien au-delà du développement côté serveur. Ces simplifications, rendues largement possibles par Java 5, seront appréciées par les développeurs qui avaient choisi les EJB pour leurs capacités de montée en charge et de transaction, mais à qui on a trop longtemps tenu le discours des outils de développement comme unique réponse pour simplifier leur travail.

Java EE 5 et EJB 3 sont disponibles depuis mai 2006 après trois années de travail et de nombreuses contributions de la communauté Java. Le serveur d'applications GlassFish et l'IDE NetBeans, tout deux Open Source, implémentent dès à présent la totalité de ces nouvelles spécifications. Ce paysage idyllique ne serait pas complet sans un livre en français sur les EJB 3 qui est là pour que vous n'ayez pas à vous fier au brouhaha (auquel je contribue), ni à une hypothétique règle des versions 3.0.

J'espère que le travail itératif et passionné de l'équipe SUPINFO sur cet ouvrage en fera une référence et que vous constaterez comme moi une véritable synergie entre l'enthousiasme de ses auteurs et le potentiel de cette technologie EJB 3.0.

Alexis MOUSSINE-POUCHKINE
Architecte Java de Sun Microsystems France
Meudon, le 30 août 2006.

Table des matières

Avant-propos

Depuis sa création, le langage Java a toujours été synonyme de portabilité. Insatisfait par les langages C et C++ utilisés en 1990 chez Sun Microsystems, Patrick Naughton se vit proposé de travailler sur une nouvelle technologie : le projet *Stealth* (furtif). Rebaptisé *Green Project* avec l'arrivée de James Gosling et de Mike Sheridan, cette petite équipe envisage alors d'élaborer une nouvelle technologie basée sur le C++ et permettant d'offrir à Sun une perspective unique : la portabilité.

Cette portabilité ayant pour but d'être poussée à l'extrême et de permettre de développer avec le même langage tout type d'appareil, le langage C++ est très vite abandonné. Les ressources qu'il demandait, au niveau de la gestion mémoire et des erreurs, et les problèmes de sécurité en sont les principales causes. Un nouveau langage est alors mis en chantier pour permettre cette portabilité : *Oak* (chêne). Java était né.

Dès 1992, les trois ingénieurs présentent un des premiers PDA, le *Star7*, incluant le système d'exploitation *Green* et développé grâce au langage *Oak*. Celui-ci comprenait une interface graphique accompagnée d'un agent intelligent nommé *Duke*, qui deviendra, par la suite, la mascotte de Java. *Green*, bien que prometteur, est dépassé trop rapidement par Silicon Graphic, et disparaît. *Oak*, quant à lui, est recentré vers le développement web. Il est alors rebaptisé *Java*.

Beaucoup diront que Java est un acronyme signifiant « **J**ames Gosling, **A**rthur **V**an Hoff and **A**ndy Bechtolsheim » ou bien « **J**ust **A**nother **V**ague **A**cronym » (juste un autre vague acronyme). D'autres, encore, prétendront que ce langage porte le même nom qu'une marque de café, à cause des quatre premiers octets des fichiers Java précompilés qui donnent en hexadécimal « 0xCAFEBABE ». D'après James Gosling, lui-même, ça ne serait qu'un nom choisit parmi une liste de mots créés aléatoirement.

En 1994, la plate-forme Java 1.0 alpha est alors présentée par Sun, et disponible en téléchargement. Le navigateur *HotJava* est ensuite annoncé le 23 mai à la conférence *SunWorld*, suivi de peu par l'annonce de Netscape, précisant le support de Java dans leur navigateur. Le 9 janvier 1996, Sun Microsystems organise le développe-

ment de ce langage autour du groupe *Javasoft*. La communauté Java était née et la première version de Java disponible.

En évolution constante, depuis sa première version stable du 23 janvier 1996, la plate-forme Java a connu un succès majeur au niveau des développeurs du monde entier, puis des différentes entreprises. Cet engouement pour ce langage portable n'a fait que renforcer sa communauté et a permis la mise en place du JCP (*Java Community Process*), un organisme de contrôle et de validation des évolutions de Java.

Aujourd'hui, découpé en trois plates-formes spécifiques, *Java Standard Edition*, *Java Enterprise Edition* et *Java Mobile Edition*, la technologie Java a réussi à toujours garder son objectif de base : la portabilité. C'est d'ailleurs cet avantage majeur qui a fait son succès dans le monde du périphérique mobile et de la téléphonie. Et c'est bien évidemment sa robustesse, sa sécurité et sa fiabilité qui en ont fait un standard de développement en entreprise.

Pourquoi choisir la plate-forme Java ?

Avec une stabilité, une sécurité et une robustesse depuis longtemps reconnues, la plate-forme Java est devenue un des standards de développement d'applications orientées entreprise, tout comme d'applications grand public. Les raisons motivant les développeurs à développer en Java sont multiples et varient en fonction des besoins de chacun :

- **La portabilité** : souvent décriée au début par la lenteur de la machine virtuelle Java, le principe de portabilité du Java est devenu, depuis déjà un moment, un de ses atouts majeurs. Des machines virtuelles existent aujourd'hui pour la plupart des systèmes d'exploitation, et bien d'autres périphériques mobiles.
- **Une communauté importante** : ouverte, dès ses débuts, aux développeurs, la technologie Java regroupe aujourd'hui une des plus grandes communautés. Elle produit aujourd'hui un nombre impressionnant de *frameworks* supplémentaires. Mis à la disposition de tous, ceux-ci deviennent souvent des standards. Cette communauté permet, de plus, d'accéder à des centaines de forums où il est possible d'obtenir une aide ou de trouver une solution à un problème technique.
- **Un système économique** : dans un contexte d'exploitation massive de licences, le coût des plates-formes de développement et des outils associés devient problématique pour de nombreux développeurs et surtout de nombreuses entreprises. Devant le prix très élevé de nombreux logiciels et les performances de certains sharewares, les utilisateurs se tournent vers le monde Open Source où de nombreux logiciels peuvent rivaliser tant sur le plan des performances que sur le plan de l'ergonomie.
- **Des formats ouverts** : alors que certains éditeurs s'enferment dans des formats propriétaires qui obligent à toujours utiliser des produits bien spécifiques, compatibles et souvent chers, Sun Microsystems et la communauté Java utilisent des formats ouverts et standardisés permettant une meilleure communication entre les logiciels et les systèmes.

À propos de l'ouvrage

Cet ouvrage est fondé sur les cours dispensés par le laboratoire SUPINFO des technologies Sun à SUPINFO. Son objectif est de présenter et d'illustrer la nouveauté majeure de la dernière version de la plate-forme Java Entreprise : EJB 3. Il conviendra en particulier :

- aux développeurs Java désireux de s'initier aux systèmes de persistance des données ;
- aux développeurs EJB 2 souhaitant évoluer vers un système plus modulable et plus rapide à mettre en place ;
- aux développeurs J2EE cherchant à la fois un système performant de persistance de données et un guide pratique proposant de nombreux exemples de code prêts à l'emploi.

Sur toute la longueur de l'ouvrage, nous étudierons les concepts généraux que nous appliquerons brièvement à EJB 2, puis de manière détaillée sur EJB 3. Nous nous appuierons sur une application complète illustrant les différents concepts :

- Le chapitre 1 présente les concepts principaux de l'architecture des applications d'entreprise.
- Le chapitre 2 présente l'application directe des concepts architecturaux au sein de la plate-forme Java Entreprise et les principes généraux du système EJB 3.
- Le chapitre 3 est un guide de développement des services métier d'une application entreprise.
- Le chapitre 4 est un guide de développement des objets métiers d'une application entreprise.
- Le chapitre 5 est un guide de développement d'un système de messagerie interne à une application entreprise.
- Le chapitre 6 plonge au cœur du système du gestionnaire de persistance pour en détailler le fonctionnement et présenter les optimisations possibles.
- Le chapitre 7 est une présentation des différentes options disponibles sur la récupération des objets métiers persistants. Le langage de requêtes spécifiques à EJB y sera étudié.
- Le chapitre 8 détaille les possibilités disponibles pour le développement de la couche présentation ou de l'application cliente.
- Le chapitre 9 plonge au cœur du système des transactions nécessaires pour rendre persistant les objets métiers.
- Le chapitre 10 est une présentation rapide d'environnements de développement et de serveurs d'applications indispensables aux développeurs Java Entreprise. Il y sera également présenté un *framework* qui utilise déjà la technologie EJB 3.

- Le chapitre 11 est une étude pratique complète basée sur un cas concret de développement d'une application entreprise.

> Les codes sources complets des applications utilisées tout au long de cet ouvrage sont disponibles en téléchargement sur *www.labo-sun.com*.

À propos des auteurs

SUPINFO (également appelé École supérieure d'informatique de Paris) est un établissement d'enseignement supérieur créé en 1965, reconnu par l'État en 1972, qui forme des ingénieurs informaticiens de haut niveau en délivrant un diplôme homologué par l'État de niveau 1. L'école a établi des partenariats pédagogiques avec les plus grands noms du secteur informatique mondial comme Microsoft, Oracle, IBM, Cisco Systems, Sun Microsystems, Mandriva ou encore Apple Computer.

Elle a créé des laboratoires pédagogiques francophones dont les sites Internet sont devenus des références pour le monde de l'éducation et les entreprises, et dont voici les adresses : www.labo-microsoft.com, www.labo-cisco.com, www.labo-oracle.com, www.labo-sun.com, www.labo-linux.org, www.labo-apple.com, www.labo-ibm.com.

Ces portails ouverts sans restriction à toute la communauté Internet proposent l'ensemble des travaux produits par les étudiants et enseignants impliqués. On peut y retrouver l'actualité concernant une technologie particulière, de nombreux articles, l'intégralité des cours techniques dispensés à SUPINFO ainsi que des moteurs de préparation aux certifications professionnelles.

Le laboratoire SUPINFO des technologies Sun fut créé suite à la volonté de nombreux étudiants de développer et de structurer les connaissances autour des technologies Java et Solaris. La mission du laboratoire est d'assurer un transfert de compétences vers tous les étudiants de SUPINFO, à travers le monde et est responsable de l'enseignement des technologies Sun au sein de SUPINFO.

Au travers de son portail *www.labo-sun.com* et de différentes contributions, le laboratoire est aujourd'hui une référence pédagogique reconnue.

Remerciements

Nous tenons à remercier tout particulièrement les personnes qui nous ont aidés tout au long de la rédaction de cet ouvrage, en particulier :

- Alick MOURIESSE, Arnaud LECUCQ, Olivier COMES, Erwan GUILLEMOT, Eric ROBIN, Marc PYBOURDIN et toute l'équipe de SUPINFO.
- Jean-Remi LECQ et Renaud ROUSTANG qui ont créé le laboratoire Sun.
- Toute l'équipe du laboratoire SUPINFO des technologies Sun.

- Alexis MOUSSINE-POUCHKINE, architecte Java de Sun Microsystems France, et Annick BOEL, architecte Java, pour leurs critiques et leurs validations techniques.
- Karine CZERNIEWICZ, de Sun Microsystems, pour son soutien et son encouragement.
- Toute l'équipe des éditions Dunod pour leur sympathie, leurs encouragements et leur professionnalisme.
- Stéphane ARNAULT, Hélène SEMERE, Jordane CAU, Emmanuel MACE, Jean-Michel CORGERON, Linda KAZMA, Aurélie BONNIVERT, Jessica CONTENSOUS, Jie ZHOU pour leur soutien et leurs relectures.

1

Concepts architecturaux

Objectif

Le développement d'applications est devenu, de nos jours, une partie importante des petites comme des grandes entreprises. Ayant toujours de plus en plus d'importance dans l'infrastructure même de ces entreprises, des concepts et des procédures ont dû être mis en place pour assurer un développement performant. Cette étude préalable au développement est appelée « Architecture logicielle ».

Nous étudierons, dans ce chapitre, les différents concepts d'architecture logicielle, pour vous permettre d'appréhender le développement d'applications « orientées entreprise » de la meilleure manière.

1.1 HISTORIQUE

Les concepts d'architecture ont subi de nombreuses évolutions depuis les premiers développements d'applications.

Les premières applications étaient composées d'une seule « pièce », pour ne pas dire d'une seule fonction linéaire et totalement séquentielle. Si cette solution avait l'avantage d'être performante, elle avait aussi de nombreux inconvénients : évolution difficile, maintenance lourde, partage difficile des données…

Pour éviter ces désagréments, de nouveaux systèmes ont vu le jour. Les bases de données ont tout d'abord simplifié le partage et l'échange de données. L'arrivée d'Internet et des réseaux d'entreprise ont ensuite permis l'utilisation d'applications clientes plus génériques (les navigateurs web, par exemple). Leurs architectures ont dû ainsi évoluer, afin de séparer au mieux les différentes parties de ces applications. On a alors assisté au succès du modèle *3-tiers* (3 parties) qui s'est finalement généralisé en un modèle *n-tiers*, découpant l'application en *n* parties.

L'industrialisation des développements et la taille grandissante des équipes ont aussi participé à l'évolution de ce nouveau modèle de découpage. Les développeurs, souhaitant accélérer leurs développements, ont dû factoriser leur travail pour ne pas avoir à toujours « tout recréer » à chaque projet. Ils ont, pour cela, développé des « briques » réutilisables et interconnectables facilement, réduisant, ainsi, le temps et le coût des développements.

1.1.1 Application monolithique et application client/serveur

Une application monolithique est un programme constitué d'un seul bloc et s'exécutant sur une seule machine. Ces applications sont généralement utilisées dans le domaine du temps réel ou bien au sein d'applications demandant de grandes performances. Elles restent également omniprésentes dans le monde du grand public, étant utilisées en *standalone*[1] sur les machines personnelles.

Dès l'apparition des réseaux, ces applications ont cherché à évoluer et ont abouti à des architectures dites « client/serveur », permettant de séparer la partie cliente et de regrouper la partie applicative sur un serveur. Cependant, le développement de ce genre d'application nécessite la création d'un protocole de communication entre le client et le serveur. Ce protocole étant souvent propriétaire, l'évolution de ces applications doit souvent être faite par les créateurs…

Pour les systèmes d'information d'entreprise ces solutions restent trop limitées. En effet, leur problème majeur est leur manque de séparation entre les différents éléments qui le constituent. C'est également le manque de standards qui a poussé la communauté au concept de « brique » et de séparation des tiers afin d'optimiser leurs développements.

1.1.2 Application multi-tiers

Dans le milieu professionnel, les applications doivent être plus robustes et travaillent généralement sur de gros volumes de données. Elles doivent, de plus, connecter différents départements au sein même d'une entreprise, voire avec d'autres entreprises.

La maintenance et la stabilité de ces applications sont donc des priorités pour les architectes et les développeurs. Différents modèles existent. Le plus connu est sans doute le modèle *3-tiers*, largement utilisé par les grandes entreprises ayant besoin de systèmes complexes basées sur la même organisation des informations : la logique métier. Ce modèle permettant donc d'avoir plusieurs applications différentes avec une même logique métier, elles peuvent alors mettre en place facilement des applications « distribuées » dans un environnement hétérogène[2].

---

1. *Standalone* : de manière autonome.

2. Environnement hétérogène : environnement logiciel et/ou matériel de différentes natures et/ou versions (par exemple, un parc de machines sous les systèmes Windows, Linux, Solaris, Mac OS X…).

De manière théorique, une application distribuée est une application découpée en plusieurs unités. Chaque unité peut être placée sur une machine différente, s'exécuter sur un système différent et être écrite dans un langage différent.

Le modèle 3-tiers

Ce modèle est une évolution du modèle d'application client/serveur. L'architecture *3-tiers* est divisée en trois niveaux :

- Tiers client qui correspond à la machine sur laquelle l'application cliente est exécutée.
- Tiers métier qui correspond à la machine sur laquelle l'application centrale est exécutée.
- Tiers accès aux données qui correspond à la machine gérant le stockage des données.

Chaque tiers doit être indépendant des autres et peut, par conséquent, être remplacé sans engendrer des modifications dans les autres tiers.

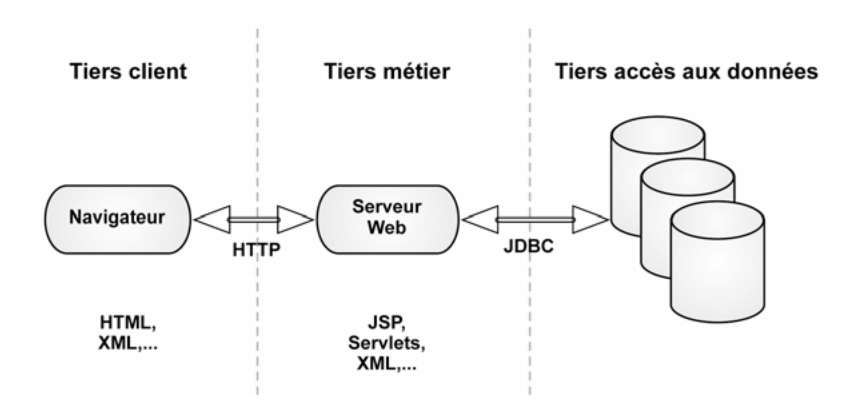


Figure 1.1 — Exemple d'architecture *3-tiers*

Ce système (fig. 1.1) utilise un navigateur pour représenter l'application sur la machine cliente, un serveur web pour la gestion de la logique de l'application et un serveur de base de données pour le stockage des données. La communication entre le client et le serveur web peut s'effectuer *via* le protocole HTTP, la communication avec la base de données *via* l'API JDBC (dans le cadre d'une application Java).

La séparation entre le client, l'application et le stockage, est le principal atout de ce modèle. Toutefois, dans des architectures qui demandent de nombreuses ressources (plusieurs bases de données, clustering, interconnexion avec d'autres systèmes, différents clients…), il sera assez limité. En effet, aucune séparation n'est faite au sein même de l'application, qui gère aussi bien la logique métier que la logique fonctionnelle et l'accès aux données.

Le modèle n-tiers

Dans le cadre d'applications beaucoup plus importantes, l'architecture *3-tiers* montre ses limites. L'architecture *n-tiers* (fig. 1.2) est simplement une généralisation du modèle précédent qui prend en compte l'évolutivité du système et évite les inconvénients de l'architecture *3-tiers* vus précédemment.

Dans la pratique, on travaille généralement avec un tiers permettant de regrouper la logique métier de l'entreprise. Ce tiers peut alors être « appelé » par différentes applications clientes.

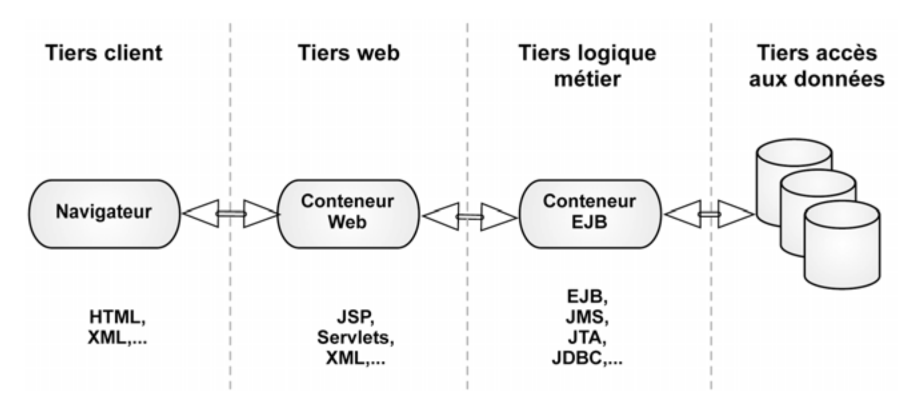


Figure 1.2 — Exemple d'architecture de type *n-tiers* en Java EE

De nombreux serveurs peuvent intervenir dans un processus (ERP, serveur de messagerie inter-application...). Si l'architecture est bien étudiée dès le début et s'exécute sur une plate-forme stable et évolutive, le développeur n'aura alors plus qu'à connecter les différents systèmes entre eux. De même, les types de clients peuvent être plus variés et évoluer sans pour autant avoir d'impact sur le cœur du système.

L'interconnexion entre ces tiers est sans doute le grand problème de cette architecture. C'est à ce niveau qu'interviennent des *frameworks*[1], comme Java EE, qui offrent un ensemble de composants standards permettant d'interconnecter les différentes technologies sans difficulté.

Les avantages de ce type d'architecture d'applications sont nombreux, on retrouve notamment :

- structure propre,
- modularité,
- facilité d'évolution,
- factorisation de code,
- utilisation de *frameworks*,

1. *Framework* : plate-forme fournissant des outils et des librairies nécessaires aux développements.

- utilisation de composants génériques,
- gain de temps,
- performances accrues…

L'inconvénient principal est, cependant, la difficulté à maîtriser cette architecture complexe. Cette phase demande une très bonne connaissance des outils disponibles sur le marché mais également une très bonne aptitude à utiliser l'abstraction et les concepts objet.

1.1.3 Application en couches

Même s'il est intéressant de séparer les traitements sur différents serveurs, cela ne règle pas tout. La maintenance et l'évolution d'une application seront facilitées si l'architecture logicielle de base est bien conçue. Contrairement à l'architecture physique qui est au niveau matériel, l'architecture logicielle se situe au niveau même de l'application exécutée dans un *tiers* : on parle alors de *couches applicatives*. L'architecte doit y séparer les traitements et les regrouper dans des ensembles.

Une couche est un sous-système possédant un comportement et des interfaces, et non un simple groupement d'entités. Le principe de couche n'est pas uniquement lié au développement mais se retrouve également en réseau avec le modèle OSI[1]. L'avantage majeur du découpage en couches est de pouvoir séparer les traitements, réduisant alors leur couplage[2]. L'évolution et le maintien de l'application en sont alors facilités.

Généralement, la couche « Persistance » permet d'accéder aux données, la couche « Services » gère les règles métiers, la couche « Présentation » gère le rendu des données… Il ne faut cependant pas confondre les couches et les outils. En effet, les outils sont indépendants de l'application alors que l'implémentation des couches est généralement étroitement liée à celle-ci. L'organisation de ces couches et leur interconnexion sont très souvent basées sur les *design patterns*[3] et les bonnes pratiques de développement.

Bien que la plupart des architectures en couches essayent de respecter le même principe de communication que le modèle OSI, il n'est pas rare que cette règle n'y soit que rarement appliquée entièrement. La norme définit plutôt une architecture lâche ou transparente dans laquelle les éléments d'une couche collaborent ou sont couplés avec plusieurs autres couches.

L'accès d'une couche supérieure à une couche inférieure doit être traité de façon rigoureuse afin de ne pas coupler les deux couches de façon forte. Il est cependant

1. Modèle OSI : base commune à la description du processus de fonctionnement de tout réseau informatique (http://fr.wikipedia.org/wiki/Modèle_OSI).
2. Couplage : mesure abstraite de dépendance entre des éléments liés (http://www.labo-sun.com/resource-fr-essentiels-833-5-architecture-j2ee.htm#h3n1).
3. *Design pattern* : modèle de conception destiné à résoudre les problèmes récurrents suivant le paradigme objet (http://fr.wikipedia.org/wiki/Design_pattern).

inutile de passer du temps à essayer de supprimer tous les couplages, certains n'étant pas forcément gênants dans le développement ou l'évolution future de l'application. Est-il justifié de gaspiller du temps à tenter de rendre abstrait ou de masquer un élément qui a peu de risques de changer, ou dont le changement aurait un impact négligeable ?

Le couplage est donc un principe à prendre très au sérieux dans tout développement. Il est important de bien étudier les différentes couches et leurs couplages pour ne rendre abstrait que les éléments nécessaires.

1.2 DÉTAILS DES COUCHES

L'architecture logicielle d'une application est, dans de nombreux cas, constituée des couches suivantes (fig. 1.3) :

- présentation (client),
- application,
- métier, ou logique business,
- accès aux données.

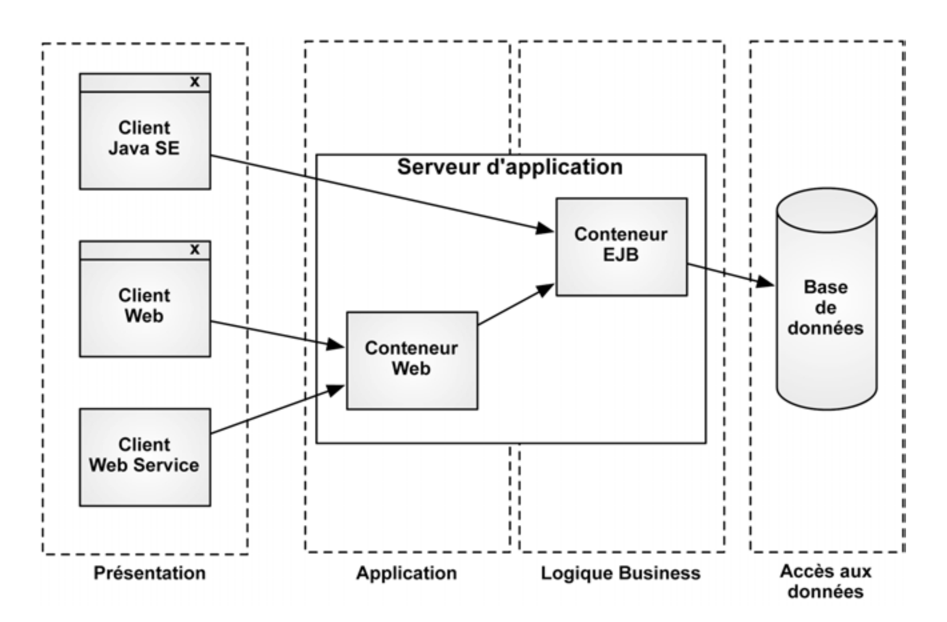


Figure 1.3 — Architecture générale d'une application en couches (Java EE)

Il est bien entendu possible d'ajouter des couches, si le besoin s'en fait sentir. Si l'application doit être accessible à partir d'un protocole réseau particulier afin de recevoir des alertes, il est alors probable qu'une couche réseau soit ajoutée.

L'ordre de présentation des couches est lié à l'ordre dans lequel elles sont exécutées lors d'une requête de l'application cliente.

1.2.1 Couche présentation

La *couche présentation* est liée au type de clients utilisés. Si vous souhaitez travailler avec un client riche, vous devrez utiliser l'ensemble des outils et des librairies mis à disposition pour ce type de client (plus particulièrement J2SE).

Dans une architecture Java EE, on utilise principalement des applications web pour l'interface utilisateur. En effet, lorsque votre application n'a aucunement besoin d'utiliser des composants graphiques très spécifiques, il est bien plus pratique d'utiliser les possibilités d'un navigateur HTML que de développer spécialement une application graphique.

Dans le cas d'une application web, l'application cliente doit interpréter le HTML envoyé. La génération de celui-ci se situe donc au niveau du serveur d'applications. En Java EE, les composants utilisés par le développeur sont les JSP (*JavaServer Pages*) et les TagLib[1]. Cette couche est également nommée Vue, dans le concept du MVC (Modèle Vue Contrôleur)[2].

La couche présentation n'est cependant pas toujours synonyme de graphique. La couche présentation d'un service web, par exemple, est la représentation XML des données retournées par celui-ci.

1.2.2 Couche application

La *couche application* sert de médiateur entre la couche présentation et la couche métier, et contrôle l'enchaînement des tâches. Elle est chargée de connaître et de gérer l'état (connecté, en attente, déconnecté…) des sessions des clients connectés, si l'état est de type « conversationnel ». Cette couche représente le contrôleur dans un modèle MVC.

Le composant de base utilisé dans un environnement Java EE est la *servlet*. Ces composants servent de point d'entrée à l'application. Ils doivent traiter la requête et faire les appels nécessaires afin de récupérer ou d'enregistrer des données. Ils redirigeront, enfin, l'application cliente vers la partie adéquate de la couche présentation.

Attention : il est difficile, au début, de comprendre le rôle des servlets sans les assimiler aux JSP. L'implémentation d'un modèle MVC correct impose que les requêtes soient d'abord traitées par le contrôleur.

1. TagLib : bibliothèques de balises JSP qui agissent comme des extensions au HTML ou au XML.
2. MVC (Modèle vue contrôleur) : modèle de conception séparant le modèle de données, l'interface utilisateur et la logique de contrôle.

Si vous souhaitez, par exemple, traiter un formulaire, vous utiliserez une servlet plutôt qu'une page JSP. Les pages JSP ne doivent comporter qu'un minimum de code Java afin d'être optimale en terme de maintenance.

Cependant les servlet/JSP étant des composants de bas niveau, elles n'offrent que des fonctionnalités de base et ne permettent pas une interconnexion simple avec les autres couches. Celles-ci peuvent cependant être utilisées au sein de *framework* fournissant ces fonctionnalités d'interconnexions. Le plus connu est sans doute Struts[1] qui met en avant l'ensemble de l'architecture MVC.

Suite à Struts, les JSF (*JavaServer Faces*) ont vu le jour et risquent de s'afficher comme un standard d'ici peu, étant désormais intégrées dans la plate-forme Java EE.

1.2.3 Couche métier

La *couche métier* est la couche principale de toute application. Elle doit s'occuper aussi bien des accès aux différentes données qu'à leurs traitements, suivant les processus définis par l'entreprise. On parle généralement de traitement métier. Cette expression regroupe :

- la vérification de la cohésion entre les données,
- l'implémentation de la logique métier de l'entreprise au niveau de l'application,
- la gestion du *workflow*[2].

Dans le domaine de l'assurance, par exemple, un traitement métier pourrait s'apparenter à une méthode de calcul d'une prime d'assurance.

Il est cependant plus propre de séparer toute la partie accès aux données de la partie traitement de la logique métier. Cela offre plusieurs avantages. Tout d'abord, les développeurs ne se perdent pas entre le code métier, qui peut parfois être complexe, et le code d'accès aux données, plutôt élémentaire mais conséquent. Cela permet aussi d'ajouter un niveau d'abstraction sur l'accès aux données et donc d'être plus modulable en cas de changements de type de stockage. Il est alors beaucoup plus facile de se répartir les différentes parties au sein d'une équipe de développement.

Dans le cas d'applications Java EE de grande envergure, cette couche est représentée par les EJB (*Entreprise JavaBeans*).

1. *Struts* : collection de librairies Java conçue (par Apache) pour construire rapidement la « charpente » d'application J2EE (http://www.labo-sun.com/resource-fr-essentiels-859-0-struts.htm).
2. *Workflow* : processus de traitement métier à plusieurs étapes.

Ces composants sont particulièrement adaptés aux panels hétérogènes de clients (Web, client riche, client d'un autre langage...) devant se connecter à une même logique métier. Si votre application n'a, cependant, besoin que d'un seul type de client, il vous sera alors possible d'étudier d'autres solutions moins « gourmandes » en ressources. Hibernate, Spring, iBatis pour n'en citer que quelques-unes.

1.3 MODÈLE EJB

Le modèle d'architecture « distribuée » impose l'idée qu'une application est découpée en plusieurs unités. Cependant, il serait quasi impossible de créer ce type d'application « *from scratch* » (à partir de rien). En effet, la seule phase de création de la plate-forme supportant ce type d'applications aurait un coût important aussi bien en termes d'argent que de temps.

Des standards ont alors vu le jour. Le plus général est sans aucun doute Corba[1] qui correspond au modèle idéal des applications distribuées. Cependant, la lourdeur et la complexité de mise en œuvre de ce genre d'applications sont les inconvénients majeurs de cette technologie. C'est pourquoi, un modèle plus restrictif mais plus performant a réussi à gagner une part de marché importante : le modèle EJB. C'est, bien entendu, celui que nous allons utiliser et décrire tout au long de cet ouvrage. Même s'il se base sur le protocole standard IIOP[2], pour l'échange de données, il reste principalement utilisé dans le monde Java.

Nous détaillerons le point essentiel de ces technologies qui est « l'objet distribué ». Puis nous nous consacrerons plus en détail à l'implémentation et l'architecture EJB.

1.3.1 Objets distribués

La communication entre les applications a été introduite par la programmation client/serveur et le principe des *socket*[3] (fig. 1.4). Ce modèle de bas niveau oblige les concepteurs et développeurs à inventer des protocoles pour faire communiquer leurs applications. Avec l'arrivée de la programmation orientée objet et de l'industrialisation de l'informatique, la communauté a souhaité développer des standards et surtout faciliter la communication inter-applications *via* des modèles de plus haut niveau.

1. Corba (*Common Object Request Broker Architecture*) : standard d'objets distribués permettant d'interconnecter différentes applications (développées avec différents langages) et de partager des objets entre celles-ci (http://fr.wikipedia.org/wiki/CORBA).

2. IIOP (*Internet Inter-ORB Protocol*) : protocole de transport utilisé pour la communication entre les objets Corba.

3. *Socket* : technologie Java permettant d'effectuer des connexions client/serveur élémentaires.

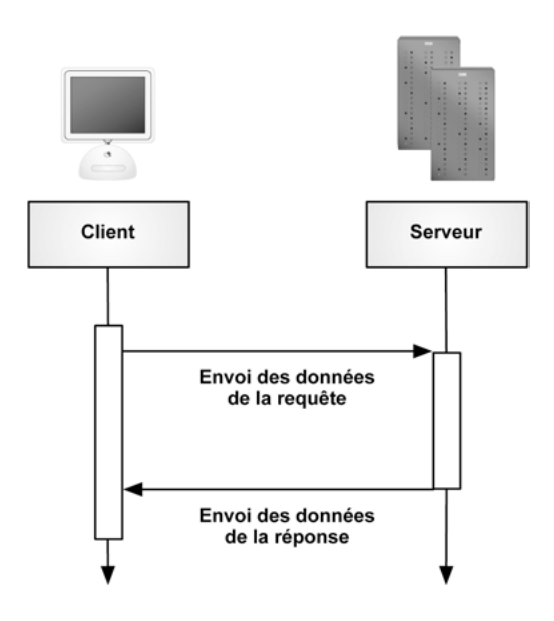


Figure 1.4 — Application client/serveur traditionnelle

Comme le montre la figure 1.4, le développement d'une application dans un modèle traditionnel oblige la création et l'implémentation d'un format de transmission propriétaire pour convertir les données d'une requête. Cette solution est donc lourde à mettre en place et a un coût. Cela ne veut cependant pas dire que ce modèle est dépassé et qu'il n'est plus exploitable.

Les objets distribués sont une solution à ce problème d'efficacité. On peut les considérer simplement comme des objets pouvant communiquer entre eux par le réseau de façon autonome. Il est souhaitable, alors, d'avoir un mécanisme permettant, au développeur de(s) application(s) cliente(s), d'effectuer un appel de méthodes sur l'objet de façon ordinaire, sans se préoccuper du format de la requête. De la même façon, le développeur de l'application serveur pourra répondre aux applications clientes, sans avoir à s'inquiéter du protocole à mettre en place.

La meilleure solution consiste à utiliser des proxy[1] qui se chargent de gérer le protocole réseau. Il faut alors un proxy serveur et un proxy client (fig. 1.5).

1. Proxy : objet créé lorsque le client active un objet distant. Il agit comme la représentation locale d'un objet distant et assure que tous les appels effectués sur le *proxy* sont acheminés vers ce dernier.

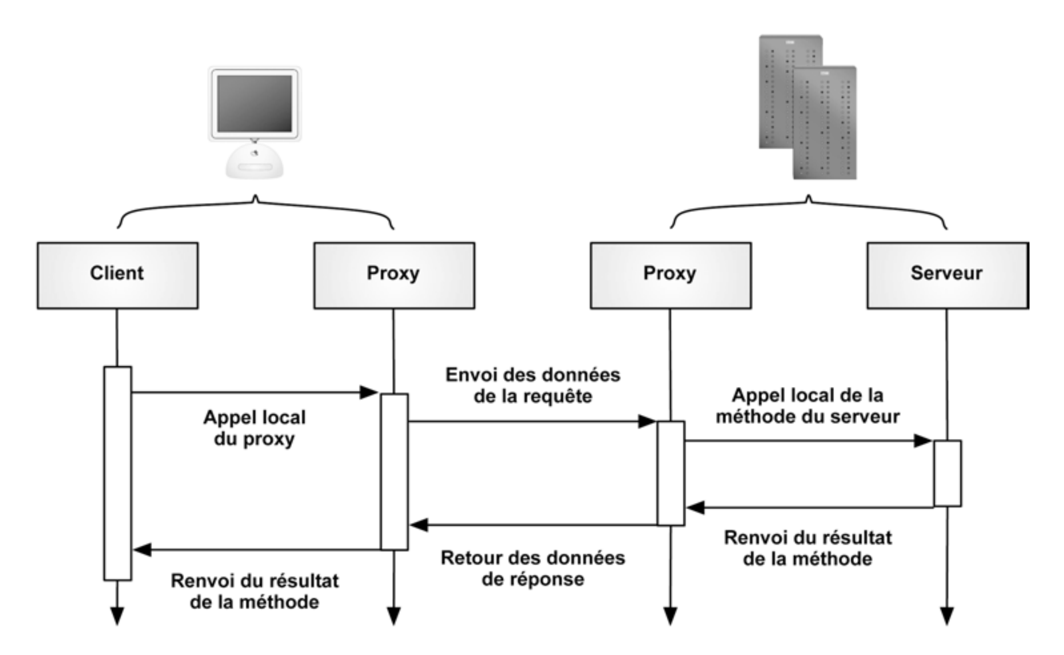


Figure 1.5 — Modèle de communication avec des proxy

Nous venons de résoudre en partie notre problème. L'approche conceptuelle est standardisée par l'utilisation du *design pattern* proxy. Toutefois, le protocole utilisé entre les proxy n'a pas été défini et nous ramène au problème de départ : la création d'un protocole de communication. L'évolution des technologies a permis le développement de standards pour l'implémentation de cette couche réseau. On retrouve :

- *Corba* : il prend en charge l'appel de méthodes entre objets distribués indépendamment de leur langage. Corba utilise le protocole Internet Inter-ORB communément appelé IIOP.
- *RMI*[1] : un Corba pour Java. C'est une implémentation spécifique à Java, basée sur le protocole IIOP.
- *SOAP*[2] : protocole utilisé par les services web. Ce protocole est indépendant du langage, étant basé sur XML. Il n'est pas lié non plus à un protocole de communication, mais est, la plupart du temps, utilisé *via* HTTP.

Même si SOAP semble une technologie d'avenir en matière d'objets distribués, elle reste beaucoup plus limitée en termes de fonctionnalités que RMI. L'implémentation des objets distribués en Java EE est réalisée par les composants EJB. Il existe une possibilité d'utiliser SOAP pour accéder à certains EJB, toutefois les applications clientes utilisent généralement RMI.

1. RMI (*Remote Method Invocation*) : cf. http://fr.wikipedia.org/wiki/RMI_%28java%29.
2. SOAP (*Simple Object Access Protocol*) : cf. http://fr.wikipedia.org/wiki/SOAP.

1.3.2 Architecture EJB

Une architecture EJB est composée d'au moins *3-tiers* : une machine cliente, une machine contenant la logique applicative et une base de données (fig. 1.6).

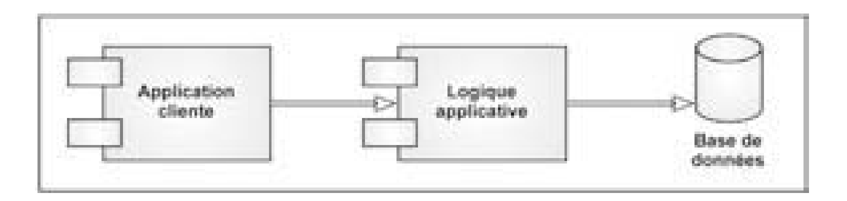


Figure 1.6 — Architecture EJB simple

Dans ce type d'architecture, les EJB se situent dans le tiers « logique applicative ». Les EJB implémentent la logique métier de l'application et représentent une abstraction de l'accès à la base de données. Cependant, l'architecture EJB est avant tout une architecture évolutive. Il est alors possible de faire évoluer une application EJB simple vers une application orientée *Business To Business* se connectant vers d'autres systèmes d'entreprise (fig. 1.7).

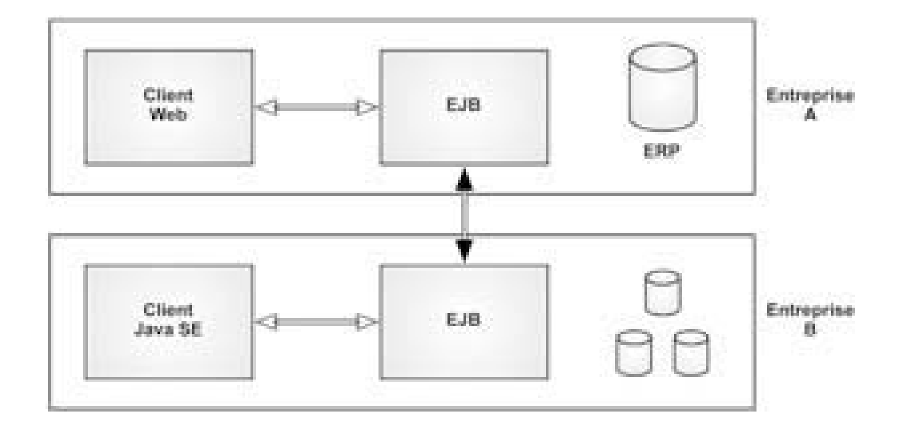


Figure 1.7 — Architecture EJB type *Business To Business*

Les EJB sont des applications exécutées côté serveur mais également englobées dans un conteneur. Chaque partie a ses propres obligations et règles, ce qui permet de faire évoluer une partie du serveur sans avoir à tout faire évoluer.

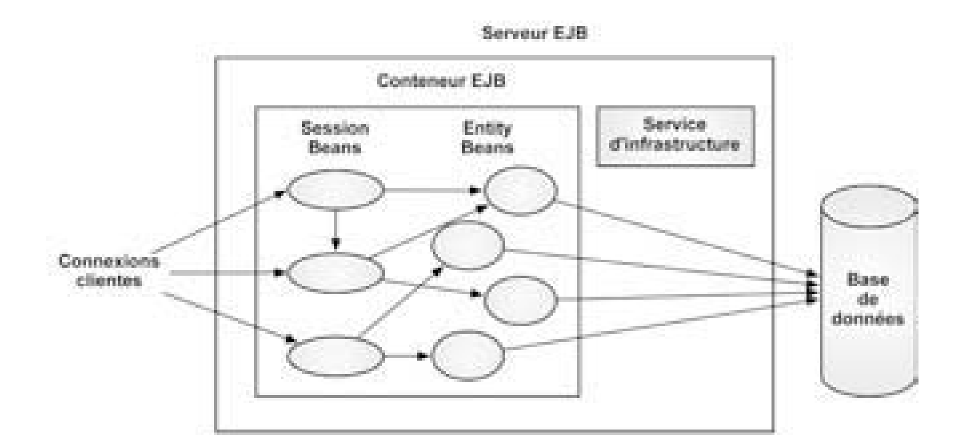


Figure 1.8 — Serveur EJB

La figure 1.8 présente la structure d'un serveur EJB. Même si le conteneur a le rôle le plus important, c'est le serveur qui aiguille l'ensemble des requêtes et gère l'ensemble des conteneurs et services. Le serveur se doit de gérer, de plus, un ensemble de services d'infrastructure communs à l'ensemble des conteneurs ou des services. La spécification Java EE (voir chapitre 2) oblige le serveur à offrir un service d'annuaire JNDI[1] et un service de transaction. Bien entendu, les serveurs d'applications fournissent généralement d'autres services facilitant le développement des applications.

De façon imagée, on peut considérer le serveur EJB comme un orchestre. Le rôle de ce serveur est celui du chef d'orchestre. C'est lui qui dirige l'ensemble des services et leur cycle de vie (démarrage, arrêt, pause…). Chaque partie de l'orchestre correspond à un service (EJB, transaction, base de données…). Ils sont tous indépendants, mais ils travaillent ensemble pour produire un résultat commun.

Le conteneur EJB

Un conteneur, c'est avant tout une « boîte noire » qui gère :

- la communication avec les services d'infrastructure,
- des applications et leur cycle de vie lié aux clients.

Le conteneur travaille de pair avec le serveur, fournissant, à eux deux, l'environnement d'exécution pour les EJB.

Le conteneur simplifie la vie du développeur grâce à une réduction des codes à réaliser, une simplification des traitements (traitements automatiques), et une main-

1. JNDI (*Java Naming and Directory Interface*) : cf. http://www.labo-sun.com/resource-FR-essentiels-646-0-JNDI.htm.

tenance réduite (séparation des couches par niveaux d'abstraction). En effet, le conteneur est à la fois une application gérant des applications mais également un outil qui génère du code pour chaque EJB.

C'est aussi le conteneur qui va permettre au développeur d'utiliser les services d'infrastructure disponibles au sein du serveur EJB (fig. 1.9), comme la connexion à la base de données, la gestion des transactions...

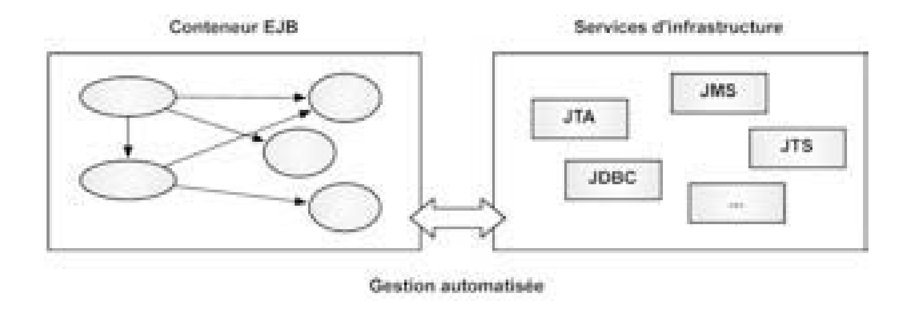


Figure 1.9 — Gestion automatisée d'un serveur EJB

Le rôle principal du conteneur EJB est, cependant, de gérer le cycle de vie des applications EJB qui lui sont associées. C'est lui qui les déploie, les stoppe et les redéploie, tout en gérant leur conformité avec les spécifications du serveur. Il permet également de contrôler les composants qui sont à sa charge.

Même si le conteneur est le point central de l'exécution des EJB, les développeurs n'ont pas à s'en préoccuper ; c'est une « boîte noire » qui, selon la spécification, s'exécute suivant certains événements bien définis.

Pour poursuivre la métaphore de l'orchestre, le conteneur est la personne qui prépare les instruments, les partitions et tout l'environnement nécessaire aux musiciens (représentés par les EJB).

Le client

Le tiers client est représenté par les applications se connectant aux EJB. Ces applications sont généralement écrites en Java, toutefois, il est également possible de se connecter à un EJB avec un client écrit dans un autre langage *via* un accès par service web (fig. 1.10).

Les clients Java utilisent généralement JNDI et RMI pour se connecter et pour appeler les méthodes des EJB. De ce fait, ils peuvent aussi bien être des applications graphiques fenêtrées (appelées clients riches), d'autres EJB, des applications internes, et, bien évidemment, des clients légers (navigateur web). Il est également possible de faire communiquer les systèmes extérieurs avec une messagerie inter-applications, comme JMS (voir chapitre 6).

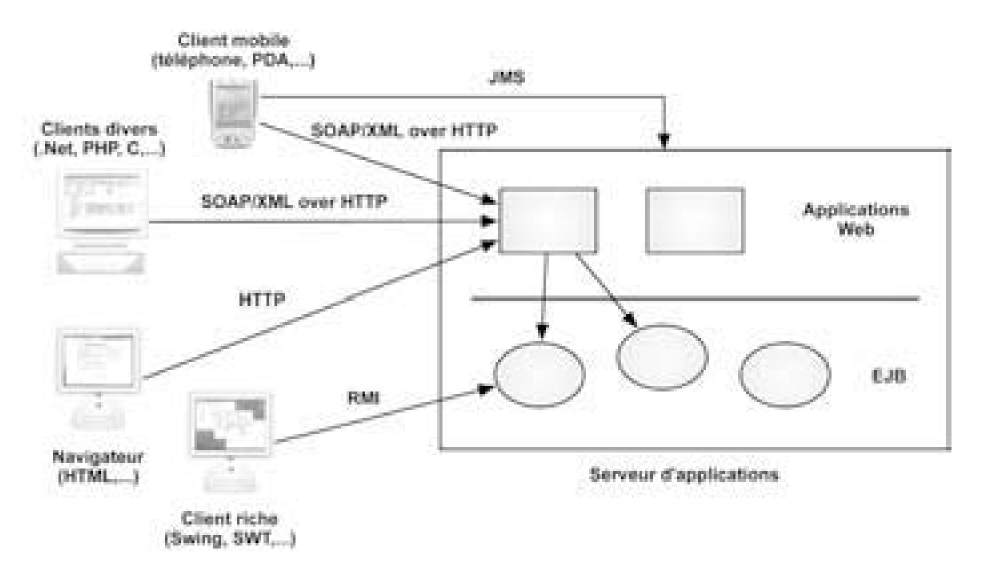


Figure 1.10 — Diversité des types de clients pour un EJB

De façon imagée, le client est le public qui écoute le concert. Le client a fait la demande pour rentrer dans la salle et a payé son ticket à la caisse (RMI), par un revendeur (HTTP, services web…) ou par petites annonces (JMS).

1.3.3 Visibilité des EJB

Au-delà de la simple délimitation des différentes couches applicatives, les EJB définissent la manière dont interagissent les différents intervenants d'une architecture Java EE.

Ils définissent, de plus, les possibilités offertes aux divers clients (application Java, applet, application s'exécutant sur le même serveur d'applications) et les modalités de communication.

Ainsi, il sera possible de définir des EJB suivant deux perspectives pour le client : une vue locale (*local*) et une vue distante (*remote*).

Visibilité locale

En adoptant une vue locale (*local*) pour un EJB, tout client exécuté dans la même machine virtuelle (autre EJB, servlet...) est en mesure d'appeler les méthodes de cet EJB. Dans cette vue, les appels de méthode de l'EJB par le client sont effectués comme dans toute application Java classique (Java SE). Les arguments sont passés par référence et il est possible, pour le client de modifier directement les objets récupérés. Il en résulte une démarcation plus faible de l'EJB vis-à-vis de ses clients. Il faut alors envisager que différents clients manient le même objet au même moment et donc anticiper les effets indésirables que cela peut induire.

En revanche, l'utilisation d'une vue locale permet d'optimiser les performances du serveur d'applications et de minimiser les ressources. Celui-ci n'a alors pas à s'occuper des spécificités liées au transport *via* le réseau (pas de sérialisation des objets, aucune communication réseau...).

Visibilité distante

En adoptant une vue distante (*remote*), un EJB met à disposition ses méthodes à des clients s'exécutant sur des machines virtuelles différentes, et donc sur des machines physiques différentes (applets, applications Java, etc.).

Dans le cadre d'une vue distante, les démarcations sont plus fortes entre un EJB et son client. Les appels de méthodes se font *via* la technologie RMI, les arguments et valeurs de retour doivent être sérialisés et ne se transmettent plus par référence. Il n'est alors plus possible qu'un client modifie le même objet d'un autre client, et il est donc plus aisé de délimiter les différents domaines de sécurité.

Par contre, l'utilisation d'une vue distante a aussi des inconvénients. Les objets devant être « transportables » à distance, le conteneur doit sérialiser/désérialiser ces objets pour les transmettre *via* le réseau. Il en résulte des temps de traitements plus élevés par rapport aux appels locaux.

Visibilité service web

Les services web se répandent de plus en plus sur Internet, parce qu'ils permettent d'utiliser n'importe quel service à partir de n'importe quel langage.

Il est possible de spécifier la visibilité de votre EJB avec le type *webservice* pour qu'il puisse être utilisé à la manière d'un service web. Toutefois, ce choix se restreint aux EJB de type *Stateless Session Bean* (voir chapitre 4). Cette technologie manque cependant de maturité et doit pour le moment rester au niveau des préoccupations de veille technologique.

Comment choisir le type d'accès ?

L'adoption d'une vue locale ou distante se décide lors de la création des EJB. Il est tout à fait possible d'envisager l'utilisation de ces deux vues simultanément, mais il faudra prendre en compte le fait qu'elles ne sont pas totalement équivalentes (notamment au niveau de la sécurité avec les passages par référence/valeur).

La différence principale entre ces deux types est sans doute l'accès local ou l'accès à distance. Si l'application cliente s'exécute au sein de la même machine virtuelle alors la vue locale sera certainement la plus appropriée et inversement pour la vue à distance. Le choix ne doit pas être générique, mais doit être fait par rapport aux besoins de l'application. Il est cependant possible de favoriser les accès locaux en plaçant un intermédiaire (le conteneur web) entre les EJB et le client distant.

La vue *webservice*, quant à elle, est à utiliser lorsque le client n'est pas écrit en Java ou lorsque vous souhaitez rendre votre service le plus ouvert possible. Typiquement, si vous souhaitez donner la possibilité à vos acheteurs de consulter votre cata-

logue de produits de n'importe quelle manière, il peut être judicieux de leur donner l'accès à un tel service web.

Pour reprendre notre comparaison à un orchestre, la vue locale serait celle d'un spectateur présent dans la salle et la vue distante, celle d'un spectateur écoutant sur sa radio la rediffusion mise à disposition par l'orchestre. La vue *webservice* serait, quant à elle, celle d'une chaîne TV externe, téléchargeant l'enregistrement du concert complet, mis à disposition par l'orchestre (*Business to Business*).

1.4 EXEMPLE D'ARCHITECTURE

Afin de faciliter les liaisons entre les différents exemples du livre, nous avons choisi de les illustrer au sein d'une même application. Voici une brève présentation de celle-ci avec les détails d'architecture qu'elle implique.

Elle se nomme « StockManager » et permet, simplement, de gérer des portefeuilles d'actions.

> **Remarque :** nous n'allons pas détailler toutes les phases de développement dans cet ouvrage, mais nous nous concentrerons sur l'aspect EJB. De même, nous ne détaillerons pas de façon exhaustive les différentes fonctionnalités et les analyses menées lors du développement pour des raisons de simplification. Le détail complet de l'application pourra être téléchargé à l'adresse suivante :
> www.labo-sun.com/resource-fr-articles-1176-0-ejb3-applications-exemples.htm.

1.4.1 Cas d'utilisation et analyse

Cette application propose à n'importe quelle personne de s'inscrire et de gérer ses portefeuilles d'actions.

L'application « StockManager » implémente l'ensemble de ces cas d'utilisations simplifiés :

- Inscription
- Authentification (Login)
- Désinscription
- Administration du compte
- Modification des informations du compte
- Ajout d'un portefeuille
- Listage les portefeuilles
- Modification du portefeuille
- Gestion des actions (acheter, vendre)
- Suppression du portefeuille

Ces fonctionnalités sont assez simples mais communes à de nombreuses applications.

L'analyse de ces besoins introduit différentes entités représentant les acteurs réels ou virtuels de cette application. Entre autres, on retrouve les entités `User`, `Portfolio`, `FinancialProduct`, `Transaction`, `Address`... Nous ne rentrerons pas dans le détail des relations entre les entités, elles sont schématisées dans la figure 1.11.

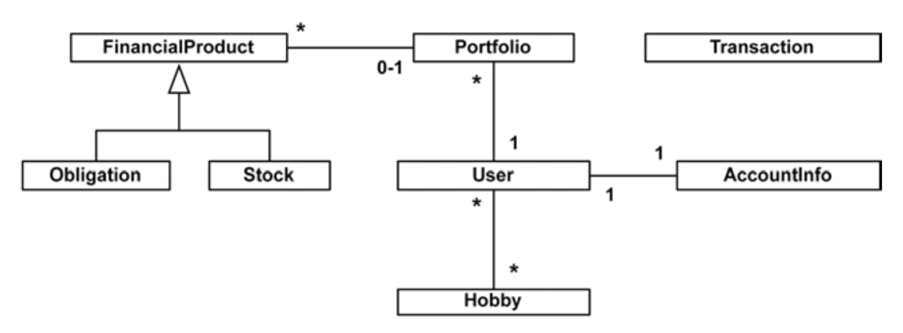


Figure 1.11 — Entités et relations

Le système propose trois types d'accès : client riche, client web et service web. Suivant le type de clients, l'utilisateur peut exécuter plus ou moins de fonctionnalités. Le client riche permet de gérer l'ensemble des fonctionnalités. L'application web permet de regarder les informations sur les portefeuilles. L'accès client par service web permet d'avoir les cours de la bourse.

1.4.2 Architecture retenue

L'application (côté serveur) s'exécute sur une unique machine rassemblant l'environnement d'exécution complet. Celui-ci comprend (fig. 1.12) :

- Un serveur d'applications (JBoss, compatible EJB 3).
- Deux serveurs de base de données (MySQL).
- Client1 de type navigateur web (HTML).
- Client2 de type riche (Java).

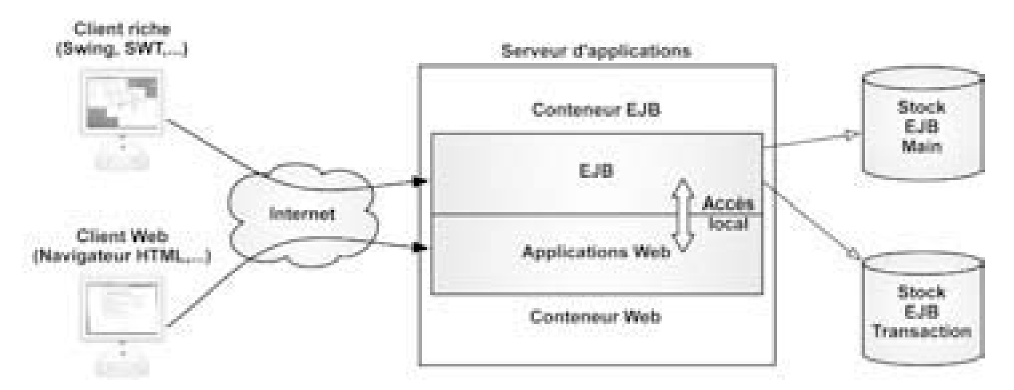


Figure 1.12 — Architecture physique

Remarque : pour des raisons pratiques, nous n'utiliserons qu'un seul serveur MySQL avec deux bases de données configurées.

Même si notre application suit essentiellement le modèle *3-tiers*, la pratique montre que l'on tente très souvent de les rassembler au sein d'une même machine. Ce qui sera le cas, ici : notre serveur d'applications sera situé sur la même machine que notre SGDB.

L'architecture matérielle de cette application reste très simple pour une application d'entreprise.

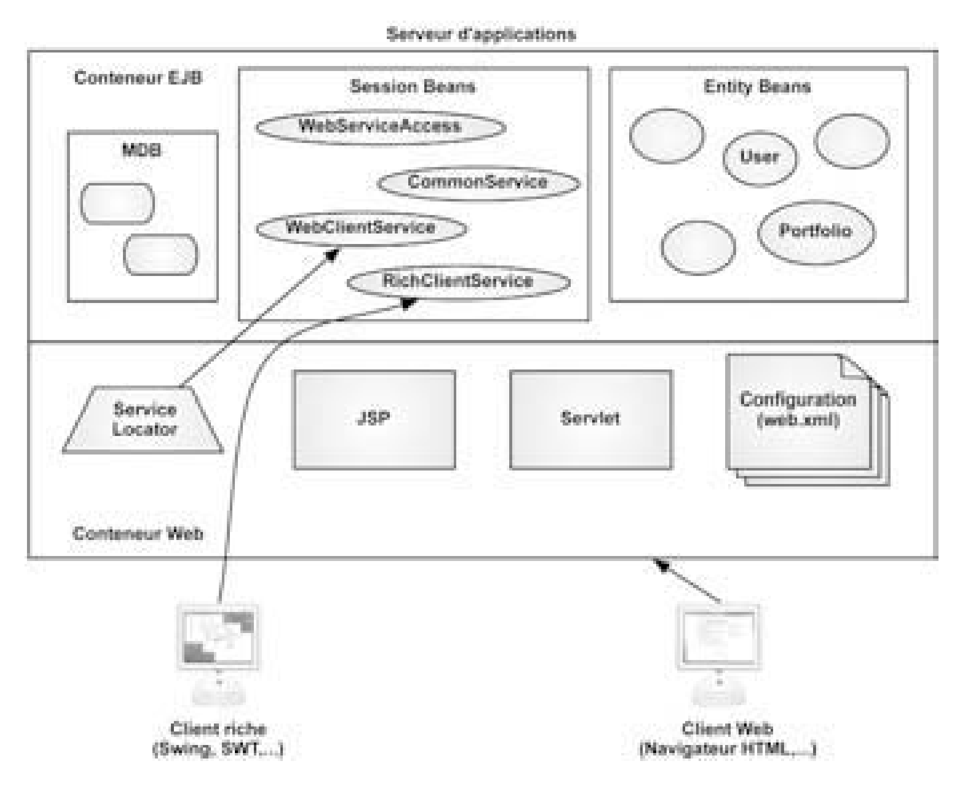


Figure 1.13 — Architecture logicielle

L'architecture logicielle (fig. 1.13) mise en œuvre pour cette application est plus qu'intéressante. Elle offre un accès commun à la logique métier pour les applications web et client riche. Cette logique est regroupée au sein des EJB `RichClientService`, `WebClientService` et `CommonService`. Comme leur nom l'indique, ils servent respectivement aux clients riches, aux clients web et le dernier regroupe les méthodes métiers communes aux clients.

Afin d'ouvrir notre application à d'autres plates-formes, nous avons également ajouté l'EJB `WebServiceAccess` qui ouvre un accès *via* les services web sur une partie très limitée de notre logique métier (possibilité de récupérer les cours de la bourse).

L'application web se situant dans le même serveur que les EJB, elle peut alors accéder à ces derniers localement. A *contrario*, les clients riches s'exécutent sur des machines virtuelles différentes, imposant l'utilisation d'un accès à distance. Cela explique donc l'utilisation d'un EJB spécifique à chacun de ces deux types de clients.

Nous détaillons, dans les chapitres suivants, les différentes technologies utilisées pour le développement de cette application, et tout particulièrement celui de la partie métier, avec EJB.

En résumé

L'architecture logicielle est aujourd'hui une partie primordiale du développement d'applications. Bien que complexe, le développement en tiers et en couches est un passage obligatoire pour tout développement d'applications orientées entreprise. Et ce, particulièrement avec les applications Java EE qui imposent le respect de ces concepts.

Il existe cependant de nombreux autres modèles bien plus complets. Depuis deux ou trois ans, on entend parler de SOA (*Service Oriented Architecture*), une évolution des concepts architecturaux, que nous ne pourrons présenter ici, pouvant être lui-même le sujet de plusieurs ouvrages.

L'exemple d'architecture vu précédemment n'est pas anodin et nous servira de base pour les différentes explications conceptuelles expliquées tout au long de ce livre.

2

Java EE 5 et les EJB 3

Objectif

L'architecture logicielle est aujourd'hui une étude obligatoire à tout développement. Tous les principes et concepts qu'elle englobe se retrouvent, en Java, dans le *framework* Java EE. *Framework* depuis longtemps reconnu, il est devenu un des standards de développements orientés entreprise les plus appréciés par la communauté.

La dernière évolution de Java EE intégrant un grand nombre de nouveautés, nous commencerons par vous les présenter. Nous nous concentrerons cependant sur la plus importante de celles-ci : EJB 3.

2.1 PRÉSENTATION DE JAVA EE 5

2.1.1 Qu'est-ce que Java EE 5 ?

Java EE 5[1], ou plus communément Java EE, est le nouveau nom donné à J2EE[2]. Ce n'est en aucun cas un nouveau langage mais une couche de haut niveau s'appuyant sur J2SE 5.0[3].

Java EE est un ensemble de composants, concepts et principes ayant pour but de fournir des services aux applications hébergées au sein d'un serveur. On parle sou-

1. Java EE 5 (*Java Enterprise Edition* 5).
2. J2EE (*Java 2 Enterprise Edition*).
3. J2SE 5.0 (*Java 2 Standard Edition* 5.0).

vent de Java EE comme étant une plate-forme de développement (ou *framework*). Sous ce terme « Java EE » se cache avant tout une spécification[1].

Java EE représente la théorie, définissant les spécifications d'un serveur d'applications compatibles ainsi que les procédures à suivre. L'implémentation du serveur est à la charge des entreprises souhaitant utiliser cette spécification. Il en existe de nombreuses, nous en présentons quelques-unes à la fin de cet ouvrage (voir chapitre 10).

Dans une application Java EE, la logique applicative est séparée en différents composants qui peuvent être installés sur différentes machines.

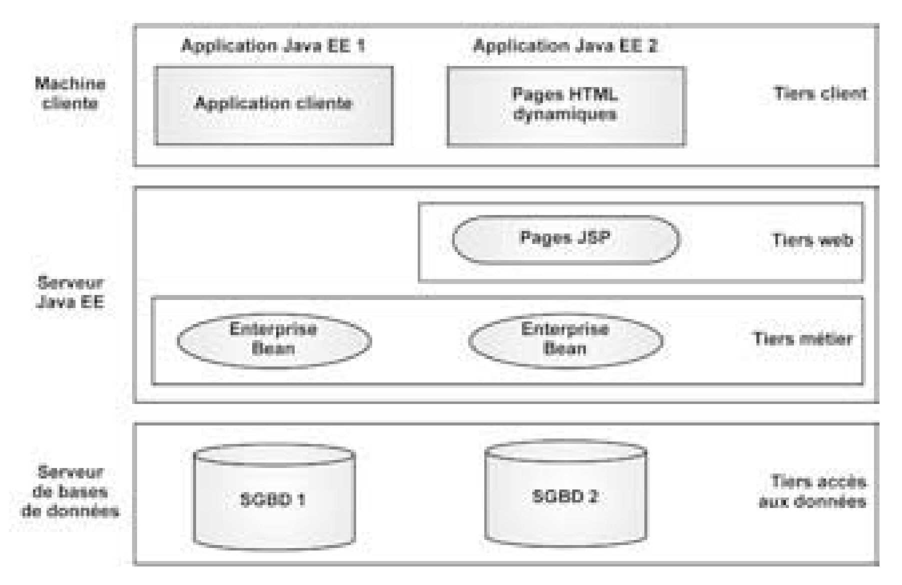


Figure 2.1 — Architecture d'applications *multi-tiers* Java EE

La figure 2.1 présente deux applications *multi-tiers*. Plusieurs machines physiques y sont utilisées :

- *Tiers client* : machine cliente (navigateur web, application client riche…)
- *Tiers web* : serveur Java EE
- *Tiers métier* : serveur Java EE
- *Tiers données* : serveur de données (SGBD, EIS[2]…)

Une application Java EE est généralement considérée comme une application *3-tiers* car elle est distribuée sur trois localités différentes : machine client, serveur Java EE et serveur de données. Elle regroupe un ensemble de composants qui peuvent interagir entre eux.

1. Spécification : description formelle de la constitution et/ou du fonctionnement d'un système.
2. EIS (*Enterprise Information System*).

> **Remarque :** habituellement, une seule machine concentre les tiers web, métier et données, pour plus de facilité de maintenance. La séparation des tiers sur des machines physiques s'effectue parfois lorsque la montée en charge d'une partie de l'application nécessite une machine à elle seule.

La spécification définit trois types de composants :

- *Client* : application cliente et applet.
- *Web* : servlet et JSP (*JavaServer Page*).
- *Métier* : EJB (*Entreprise Java Bean*).

Les différences entre une application J2SE (standard) et une application Java EE (entreprise) résident dans le fait que cette dernière regroupe un ensemble de composants, suivant la spécification EE, et doit être déployée dans un serveur d'applications EE pour y être exécutée.

La plate-forme EE utilise l'ensemble des fonctionnalités de la SE, et définit une architecture globale pour vos applications (fig. 2.2).

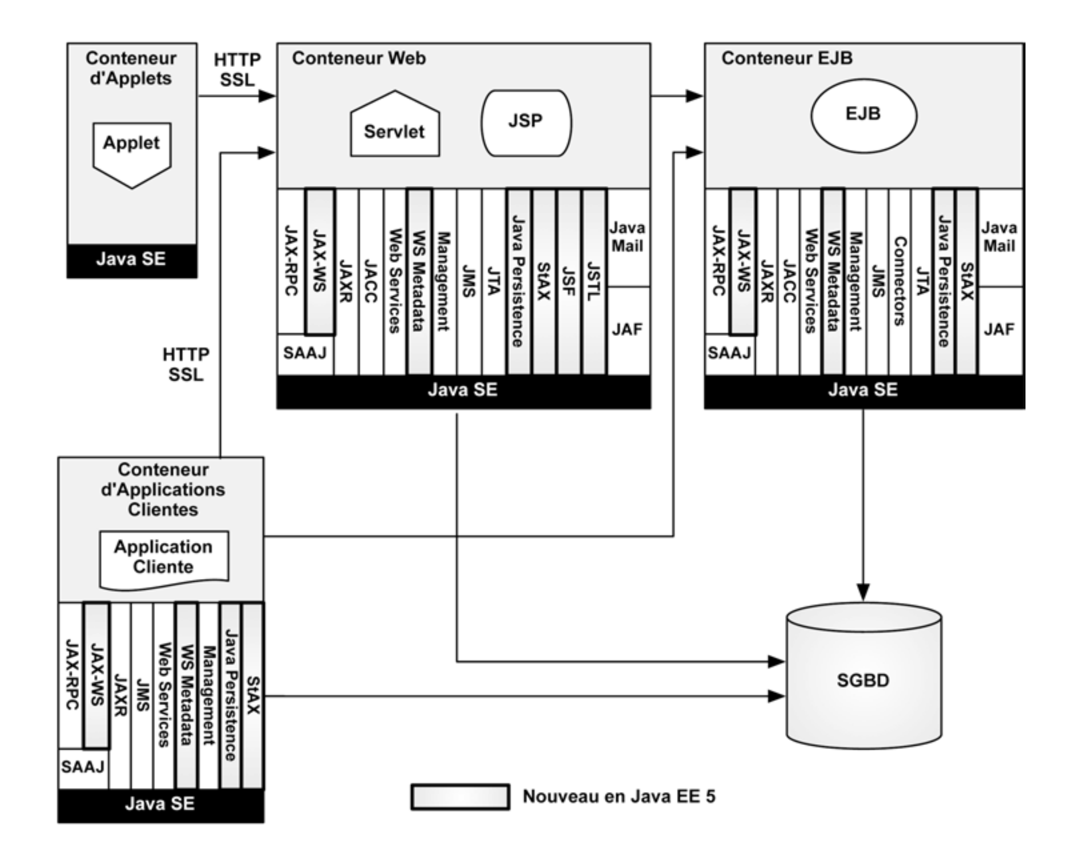


Figure 2.2 — Architecture Java EE 5

On remarque bien l'implication de Java SE au sein même de l'architecture Java EE. La localisation d'un EJB, par exemple, est réalisée avec JNDI[1] (API Java SE) et l'appel des méthodes de celui-ci *via* RMI (API Java SE).

2.1.2 D'où vient Java EE 5 ?

J2EE, à l'origine, est le résultat d'une volonté de standardiser la mode d'exécution des serveurs d'applications Java. Cette standardisation avait également pour objectif de faciliter la portabilité des applications métiers d'un serveur à un autre.

La plate-forme J2EE a été conçue par le JCP (*Java Community Process*), une communauté dirigée par Sun Microsystems, lancée en 1998.

Depuis 1999, la plate-forme évolue, et bien que très minimale au départ (les servlets et les EJB n'étaient pas encore standardisés), elle s'est améliorée et a su faire sa place sur le marché.

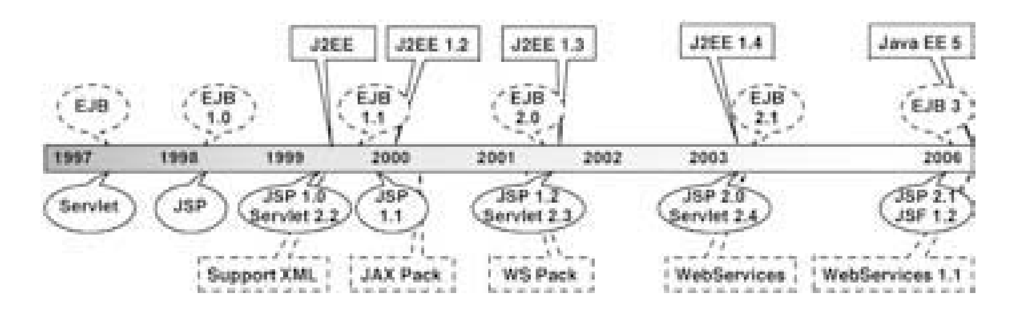


Figure 2.3 — Évolution de la plate-forme Java EE

De nombreux supports et nouvelles versions ont été intégrés entre 1997 et 2003. Suite à une certaine stagnation des évolutions de la plate-forme, une grande réflexion (environ deux ans) s'est portée sur la simplicité de celle-ci. En parallèle, les serveurs d'applications ont vu leurs performances s'accroître considérablement. La nouvelle version Java EE 5 en est née.

Le nouveau nom Java EE 5 a été décidé afin de simplifier et d'éviter toute confusion. Le « 2 » (de J2EE) a été supprimé et les numéros de versions seront, à présent, des nombres entiers. Les mises à jour s'appelleront « update » (par exemple Java EE 5 Update 1), ce qui évitera tout nombre décimal.

2.1.3 La spécification

La spécification Java EE 5 étant plus qu'importante, l'explication détaillée de celle-ci ne tiendrait pas en un seul ouvrage. Nous n'étudierons que les éléments essentiels

1. JNDI (*Java Naming Directory Interface*) : API permettant de consulter un annuaire.

à la compréhension de cet ouvrage afin d'avoir une vision globale de la spécification et des nouveautés de la version 5.

Les conteneurs

Un conteneur est un intermédiaire entre un composant et les fonctionnalités de bas niveau fournies par la plate-forme. Les services offerts par les conteneurs regroupent :

- *La sécurité* : le modèle de sécurité permet de configurer les ressources accessibles pour les utilisateurs autorisés.
- *La gestion des transactions* : le modèle transactionnel offre la possibilité de définir les relations entre les méthodes.
- *Les recherches JNDI* : fournissent une interface pour se connecter aux services de noms ou d'annuaires (LDAP, par exemple).
- *Les connexions distantes* (fig. 2.4) : le conteneur gère l'ensemble des connexions distantes entre les clients et les objets dont il a la responsabilité. Il gère également la distribution de ces objets, si nécessaire.
- *La montée en charge* : le conteneur est responsable de la bonne utilisation et du recyclage des ressources (connexion SGBD, mémoire…).

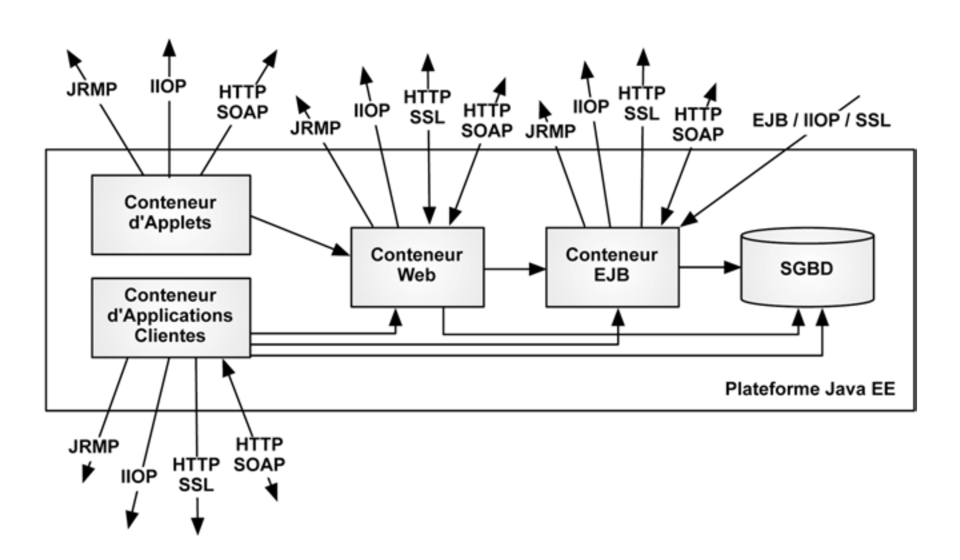


Figure 2.4 — Architecture des conteneurs

La spécification Java EE 5 n'introduit aucune nouvelle notion d'architecture, mais fournit un large panel de nouvelles possibilités permettant de faciliter le développement d'applications complexes.

Le packaging d'une application

Pour livrer une application Java EE 5 vous devez créer un fichier d'archive d'entreprise (EAR). C'est un fichier d'archive standard avec l'extension « .ear » qui regroupe un ensemble de modules EJB, Web, application cliente… comme le présente la figure 2.5.

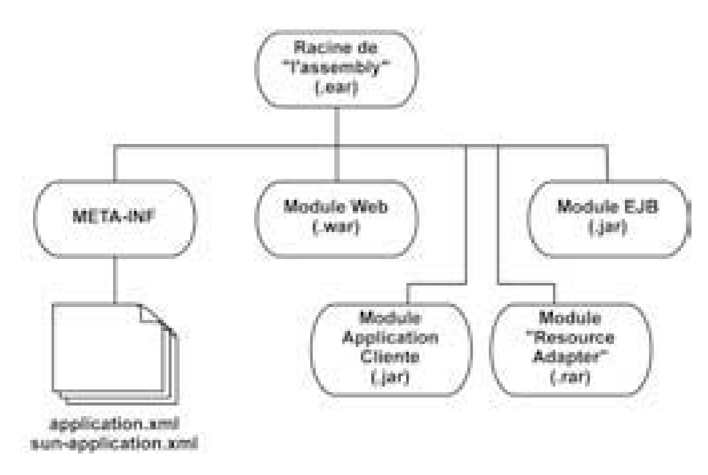


Figure 2.5 — Structure d'un fichier EAR

Chaque fichier d'archive (ear, war, jar ou rar) contient un descripteur de déploiement organisé et structuré sur le modèle XML. Celui-ci permet de définir les paramètres de déploiement d'une application (ear) ou d'un module.

Ces fichiers appelés « descripteurs de déploiement » sont liés et standardisés par la spécification Java EE 5. Il existe cependant d'autres fichiers spécifiques, liés aux serveurs d'applications. Nous retrouvons, dans la spécification, les fichiers :

- *application.xml* : il sert à déclarer l'ensemble des modules intégrés dans un fichier « .ear ».
- *ejb-jar.xml* : il configure les EJB dans un module d'un fichier « .jar ».
- *web.xml* : il configure les paramètres du module web (servlets, JSP, TagLib…) d'un fichier « .war ».

Remarque : le fichier application.xml n'est pas obligatoire pour les applications Java EE 5. Les types de modules (EJB, Web, client…) sont automatiquement reconnus lors du déploiement.

Les descripteurs de déploiement liés aux serveurs d'applications leurs sont spécifiques et ne dépendent que de ceux-ci.

- Sun Application Server utilise les fichiers sun-application.xml (ear), sun-ejb.xml (jar), sun-web.xml (war).
- JBoss utilise les fichiers jboss-application.xml (ear), jboss.xml et jbosscmp-jdbc.xml (jar), jboss-web.xml (war).

Ces fichiers permettent d'utiliser des fonctionnalités avancées du serveur d'applications qui ne sont pas intégrées aux spécifications.

Attention : l'utilisation de fonctionnalités non standard (hors spécification) entraîne des problèmes de portabilité d'un serveur d'applications vers un autre.

Les nouveautés

Les API incluses dans J2EE 1.4 sont toujours présentes dans Java EE 5. Certaines ont été améliorées, d'autres ont été ajoutées. Voici une description rapide de chacune de ces API :

- *StAX 1.0* : il permet la gestion de *streaming*[1] sur un flux XML. À l'inverse de SAX qui propose un modèle événementiel, StAX permet au développeur de gérer les étapes de la lecture du fichier XML, dans les différents conteneurs.
- *Java Persistence 1.0* : il permet la gestion de la persistance des Entity Beans dans les différents conteneurs.
- *WS Metadata 1.0* : il permet de simplifier le développement des services web du côté serveur comme du côté client. Cette API apporte un ensemble d'annotations servant à paramétrer vos services web de manière beaucoup plus simple qu'en J2EE 1.4 (WSDL, fichiers de configuration...).
- *JAX-WS 2.0* (*Java API for XML-based Web Service*) : est la pièce centrale d'un nouveau réagencement de l'API de gestion de la pile pour les services web, dans les différents conteneurs.
- *JSF 1.2* (*JavaServer Faces*) : c'est le nouveau *framework* de développement spécifique au conteneur web, celui-ci permet la création rapide d'interfaces utilisateur. Les composants principaux des JSF sont les suivants :
  - un *framework* basé sur des composants d'interface graphique,
  - un modèle flexible pour générer différents types de HTML ou tout autre langage à balises,
  - la validation des entrées (formulaires),
  - la gestion des événements,
  - un système de navigation entre les pages.
- *JSTL 1.1* (*JavaServer pages Standard Tag Library*) : il permet d'utiliser, dans les JSP, *via* le conteneur web, des éléments avancés (les TagLib) respectant la spécification.

Conseil : pour de plus amples informations et détails sur la spécification Java EE 5, n'hésitez pas à consulter celle-ci à l'adresse suivante http://java.sun.com/javaee/5/javatech.html

1. *Streaming* : principe de distribution d'un flux en continu.

2.2 OBJECTIFS DES EJB 3

2.2.1 Le problème EJB 2 : bilan

Depuis plusieurs années, une forte réaction anti-EJB 2 est apparue. En effet, les spécifications, ayant été rédigées sans attendre des retours d'expériences, les EJB 2 ont souvent été le cœur de certains problèmes qu'ils étaient censés résoudre. Voici un historique rapide du développement de la spécification EJB (cf. fig. 2.3) :

- *Mars 1998* (EJB 1.0) : une première spécification assez « légère » voit le jour. Seuls les Sessions Beans existent.
- *Novembre 1999* (EJB 1.1) : des fonctionnalités comme la sécurité, les Entity Beans (simpliste) sont ajoutés.
- *Août 2001* (EJB 2) : suite aux problèmes de performances, les interfaces locales ont été créées. La spécification EJB 2 intègre également la première version de l'EJB-QL (limité), les relations entre Entity Beans CMR et les Message Driven Beans.
- *Novembre 2003* (EJB 2.1) : des modifications mineures sont effectuées autour de EJB-QL (ajout de « ORDER BY » et des fonctions de groupes). L'EJB Timer Service[1] est ajouté.

Une première constatation se pose : la spécification a peu évolué entre août 2001 et novembre 2003. Les EJB 2 ont tellement fait parler d'eux, qu'ils ont terni l'image de J2EE. Même si certaines entreprises utilisaient EJB 2 pour ses points forts (gestion intégrée des transactions…), beaucoup d'autres avaient alors à l'esprit la lourdeur de ces EJB, l'assimilant à J2EE ; ce qui est loin d'être le cas.

Des problèmes récurrents sont, de plus, retournés par les développeurs :

- *Entity Bean* : la spécification est une erreur conceptuelle ; elle est trop restrictive. Le langage EJB-QL apporte, cependant, une « rustine » aux problèmes de flexibilité, mais reste trop limité.
- *Mauvaise productivité* : le développement d'un EJB nécessite beaucoup trop de dépendances (trop d'interfaces, longueur des descripteurs de déploiement…)
- *Complexité de développement* : obligation d'utiliser des outils externes (XDoclet, ejbgen…) et de mettre en place des *design patterns* (Service Locator, Value Object, Business Interface…). Il est également difficile d'effectuer des tests unitaires (exécution dans le conteneur EJB seulement).

Leur réputation ternie, les EJB furent laissés de côté, mais la communauté en profita pour développer des *frameworks* « maisons » qui se sont très vite répandus sur le

1. *EJB Timer Service* : service du conteneur EJB permettant de temporiser l'appel de certaines méthodes.

marché. Ces *frameworks*, appelés « conteneurs légers », ont l'avantage d'être plus performants qu'un conteneur lourd de type EJB 2, même s'ils ne supportent pas toute l'architecture distribuée des EJB. Des *frameworks*, comme Spring, Toplink, Hibernate, Cayenne…, ont très vite vu le jour sur ce principe.

Suite à cela, le JCP a décidé de mieux prendre en compte les avis de la communauté. La nouvelle spécification EJB 3 est donc le fruit d'un travail de rassemblement des éléments développés par la communauté. C'est également l'aboutissement d'une meilleure communication avec celle-ci, par l'intégration, au sein des spécifications, des retours d'expériences liés aux spécifications précédentes.

C'est en mai 2006, après deux ans d'étude, que la refonte complète de la spécification J2EE, et plus particulièrement des EJB, est réalisée. C'est le début de Java EE 5 et des EJB 3, tout en gardant une compatibilité ascendante sur J2EE 1.4 et EJB 2.

2.2.2 Les EJB 3 : évolution ou révolution ?

La nouvelle spécification EJB 3 se base sur l'expérience accumulée des développements J2EE précédents. La spécification a pour objectif de simplifier au maximum le développement des EJB et éviter l'ensemble de la lourde « tuyauterie » des anciennes spécifications.

Voici une liste de points qui ont été très largement améliorés :

- *Simplification du déploiement* : la configuration peut se faire *via* les annotations (*metadata*). Seul le paramétrage des contextes de persistance doit se faire *via* un fichier XML (persistence.xml).
- *Configuration par exception* : tout est paramétré par défaut pour faciliter et accélérer le développement (le nom de la table est le même que celui de l'Entity Bean, le nom des colonnes est le même que celui des propriétés, les transactions sont gérées par le conteneur…).
- *Facilité de développement* : l'application ne travaille qu'avec des objets « simples » (POJO[1]).
- *Modèle de vérification du code basé sur AVK* (*Application Verification Kit*).
- *Injection de dépendance* : l'utilisation d'annotations au sein des différents objets permet de faciliter les couplages entre les EJB, le conteneur web et le conteneur d'applications clientes.

Ces améliorations sont de réelles avancées pour les développeurs. Le gain de temps est bien présent, sans pour autant avoir d'impact sur les performances, bien au contraire.

1. POJO (*Plain Old Java Object*) : ce concept a pour but de représenter une idée avec le plus simple et le meilleur design. Il a souvent été opposé aux EJB jusqu'à la version 3, qui les utilise. C'est donc une classe la plus simple possible indépendante de tout système ou *framework*. Les POJO correspondent aux *JavaBeans*. Toutefois, on n'emploie guère plus ce nom à cause de son ambiguïté avec « Entreprise JavaBean ».

Pour la communauté, cette nouvelle spécification est plus qu'une évolution, c'est une révolution !

2.3 LES FONDEMENTS DES EJB 3

La spécification EJB 3 est le résultat d'un ensemble d'évolutions et d'innovations qui se sont installées sur le marché sans pour autant faire partie intégrante de la spécification Java EE 5. De nombreux *frameworks* et technologies ont approuvé certains principes et modèles de développement. La spécification a su prendre parti des avantages de chacune de ces innovations afin d'élaborer la meilleure structure possible pour les EJB.

2.3.1 J2SE 5.0 : de nouveaux apports

La première étape a été de faire évoluer J2SE 5.0 afin d'ajouter de nouvelles fonctionnalités au langage Java lui-même. Les principaux éléments nouveaux sont les énumérations, les annotations et les génériques.

Cet ouvrage n'étant pas dédié aux nouveautés du J2SE 5.0, nous n'exposerons donc que les principales modifications en relation avec notre sujet.

Les énumérations

Vous pouvez avoir à définir un ensemble fini de valeurs pour une donnée spécifique afin, par exemple, de préciser les états d'une commande (non validée, validée, complète, incomplète, expédiée…). Le type « enum » qui a été ajouté à J2SE 5.0 permet de définir un ensemble fini de constantes. Ce principe est repris des langages C et Delphi.

La définition d'une pseudo-énumération en J2SE 1.4 est mise en place grâce à une classe contenant l'ensemble fini de constantes, comme le montre l'exemple suivant :

```
// pseudo énumération (états d'une commande)
public final class OrderState {
  public static final int NOTVALIDATED = 0;
  public static final int VALIDATED = 1;
  public static final int COMPLETE = 2;
  public static final int NOTCOMPLETE = 3;
  public static final int SHIPPED = 4;
}
```

```
// classe Order
public class Order {
  …
  private int orderState;
  …
}
```

Ici, le principal inconvénient est que le développeur n'a aucun contrôle sur la valeur affectée à une donnée. La classe `Order` détient une propriété de type `int` qui représente l'état de la commande. Le développeur n'a aucun contrôle sur la valeur de cette propriété, elle peut être assignée correctement avec les valeurs de notre classe `OrderState` ou directement avec un entier ! Une commande peut alors se retrouver avec un état inexistant (valeur autre que 0, 1, 2, 3 ou 4) et donc provoquer une incohérence dans l'ensemble du système.

L'utilisation d'une énumération est la solution à ce problème de cohérence. La déclaration d'une énumération repose sur trois éléments : le mot clé `enum`, le nom désignant l'énumération et l'ensemble des valeurs de l'énumération, séparées par des virgules.

Voici le même exemple utilisant, ici, les énumérations de J2SE 5.0 :

```
// énumération OrderState (nouveauté J2SE 5.0)
public enum OrderState {
  VALIDATED, UNVALIDATED, COMPLETE, UNCOMPLETE, SHIPPED
}
```

```
// classe Order utilisant l'énumération
public class Order {
…
  private OrderState orderState;
…
}
```

La différence avec l'exemple précédent réside principalement dans le fait que la propriété `orderState` n'est plus de type `int` mais de type `OrderState`, qui représente l'énumération définie au-dessus. Nous ne pouvons donc pas spécifier une valeur en dehors de l'ensemble spécifié dans l'énumération.

> **Remarque :** une énumération est une classe Java qui peut aussi contenir des méthodes.

Les annotations

La notion d'annotation est une nouveauté très attendue des développeurs Java. Celle-ci répond à un besoin précis : pouvoir ajouter des données sémantiques à leur code, permettant alors de préciser la façon dont ces données doivent être traitées.

Auparavant, il fallait utiliser les commentaires Javadoc. Toutefois leur utilisation était contraignante et laborieuse, souvent couplée avec des outils annexes (comme XDoclet). Les annotations sont bien plus efficaces, intuitives, et surtout intégrées au langage. Elles offrent la possibilité d'associer des métadonnées aux packages, classes, méthodes, champs, paramètres, variables… Celles-ci ne sont, de plus, plus limitées aux listes prédéfinies dans le langage puisque vous avez la possibilité d'ajouter des annotations personnalisées.

Le langage Java définit six annotations de bases :

- `@Deprecated` qui redéfinit le commentaire Javadoc du même nom. Le compilateur lance un avertissement lorsque des membres annotés par `@Deprecated` sont trouvés.
- `@Documented` qui déclare que l'annotation doit être documentée par le biais des commentaires Javadoc ou autre. Elle spécifie que l'utilitaire de génération de documentation doit prendre en compte cette annotation.
- `@Inherit` qui déclare que l'annotation est automatiquement héritée. Si un membre est ainsi annoté, alors ses descendants le seront également.
- `@Overrides` qui déclare que la méthode marquée redéfinit la méthode ascendante. Si tel n'est pas le cas (mauvaise signature), une erreur est alors lancée à la compilation.
- `@Retention` qui déclare la politique d'utilisation de l'annotation. Elle définit la durée de vie de l'annotation (compilation, bytecode ou machine virtuelle). Les valeurs possibles sont `SOURCE` (compilation), `CLASS` (bytecode), `RUNTIME` (à l'exécution).
  Il existe également l'outil APT (*Annotation Processing Tool*) qui permet d'analyser les annotations au moment de la compilation.
- `@Target` qui déclare les possibilités d'utilisation de l'annotation (classe, package, champs...). Les valeurs possibles sont : `TYPE`, `FIELD`, `METHOD`, `PARAMETER`, `LOCAL_VARIABLE` et `PACKAGE`. Nous pouvons alors restreindre l'utilisation de l'annotation à un niveau prédéfini.

Ces annotations de base vont être utilisées, la plupart du temps, pour créer des annotations personnalisées, comme le montre l'exemple suivant :

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Action {
  String value();
}
```

Dans cet exemple de création d'annotation personnalisée :

- L'annotation est appelée `Action`.
- Elle peut être lue au moment de l'exécution grâce à l'annotation `@Retention(RUNTIME)`.
- Elle permettra d'annoter une classe, une interface, une énumération ou encore une autre annotation définie grâce à `@Target(TYPE)`.
- La méthode abstraite `value()` déclare un attribut de type `String` nommé `value`.

Remarque : il est à noter que la définition d'une annotation peut être elle-même annotée.

Voici un exemple basique d'utilisation de l'annotation créée précédemment :

```
@Action("mainAction")
// équivaut à @Action(value = "mainAction")
// car l'annotation Action ne contient qu'un seul attribut
public class Main {
  public static void main(String[] args) {
    try {
      Action mainAction = Main.class.getAnnotation(Action.class);
      System.out.println("Valeur de l'annotation : " + mainAction.value());
    } catch(NullPointerException e) {
      System.out.println("La classe Main n'est pas annotée");
    }
  }
}
```

Cet exemple permet de récupérer les informations d'une classe annotée pendant l'exécution d'un programme. Pour cela, vous devez utiliser l'introspection qui vous permet de travailler sur les informations d'une classe et d'une instance au moment de l'exécution (appel de méthodes, modifications des variables d'instance…)

Les génériques

Le principe des génériques existait déjà auparavant, sous le nom de *template* en C++. On parle également de polymorphisme paramétrique de type. Plus concrètement, les génériques permettent de définir un comportement unique, indépendamment du type d'objet utilisé au sein de ce comportement.

L'exemple type est celui des collections d'objets. En effet, les versions antérieures fournissent des `Collections` se basant sur le type `Object` (type parent de toute classe). L'utilisation de celles-ci faisait donc « perdre » le typage fort des objets.

```
List userList = new ArrayList();
userList.add(new User());
// vous pouvez ajouter n'importe quel type d'objet
userList.add(new Phone());
// transtypage obligatoire (cast)
// si l'objet retourné n'est pas un User
// alors une exception ClassCastException est lancée => erreur d'exécution
User user1 = (User) userList.get(0);
```

Avec l'utilisation des génériques, la collection sera typée suivant le type des objets contenus. L'utilisation d'une collection à fort typage permet d'éviter au maximum le risque de mauvaise manipulation des collections.

```
List<User> userList = new ArrayList<User>();
userList.add(new User());
// vous ne pouvez ajouter que des objets de type User (ou héritant de User)
userList.add(new Phone()); // => erreur de compilation
// pas besoin de transtypage (cast)
User user1 = userList.get(0);
```

Cela permet, de plus, de gagner en sécurité, en lisibilité et en robustesse. Les différentes erreurs de transtypage sont ici levées au moment de la compilation (et non au moment de l'exécution).

L'utilisation des collections génériques est un atout majeur pour les EJB 3. En effet, le conteneur peut, au moment de l'exécution, vérifier le type de la collection et donc connaître facilement les relations entre les entités avec lesquelles il travaille.

Remarque : les génériques offrent d'autres avantages et possibilités qui ne seront pas expliqués car ils ne rentrent pas dans le cadre de cet ouvrage.

2.3.2 Injection de dépendances

Le couplage des composants est un problème récurrent au sein d'une application. En effet, dès que vous devez rendre le plus indépendant possible les différents composants, l'architecture de l'application se doit d'avoir le moins de couplage possible. L'utilisation de *designs patterns* est souvent la solution, bien qu'ils ne soient pas toujours compatibles entre eux ou difficiles à mettre en place rapidement.

Prenons notre application d'exemple, l'application de gestion de comptes utilisateurs et de portefeuilles d'actions. Dans un tel développement, vous devez réaliser une architecture à faible couplage, permettant de séparer le traitement brut des données et le traitement métier. Pour cela, l'utilisation d'usines (*Factory*) qui se chargent de créer les instances des DAO[1] et des services sont de mise. La figure 2.6 en illustre le principe.

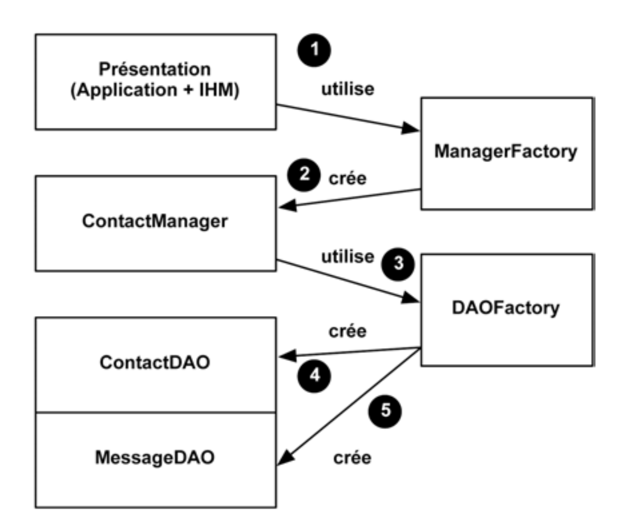


Figure 2.6 — Gestion des dépendances sans IoC

1. DAO (*Data Access Object*) : *design pattern* permettant de créer un intermédiaire entre l'application et le ou les stockage(s) de données.

Dans cet exemple, vous pouvez remarquer plusieurs particularités :

- La création des classes `ManagerFactory` et `DaoFactory` est à la charge du développeur.
- Si les dépendances sont modifiées au cours du développement, il sera impératif de modifier ces classes pour qu'elles prennent en compte ces changements.
- La gestion des dépendances devient de plus en plus complexe avec l'ampleur de l'application. L'exemple, ici, est simple (1 Service et 2 DAO), mais une application plus importante se révélera beaucoup plus complexe pour respecter le concept de modularité.

Pour remédier à ces difficultés de développement, le concept du *design pattern* « IoC » (*Inversion of Control*) a été directement intégré dans le *framework* Java EE 5. Celui-ci permet de réduire plus facilement le couplage d'une architecture grâce à un conteneur spécialisé :

- Le conteneur est responsable de la création des objets ; il effectue lui-même l'instanciation (l'opération « new »).
- Le conteneur résout les dépendances entre les objets qu'il gère.

Le conteneur joue le rôle principal dans l'application du *design pattern* IoC. C'est à lui de définir le cycle de vie des objets qu'il construit (savoir quand les construire et quand les supprimer).

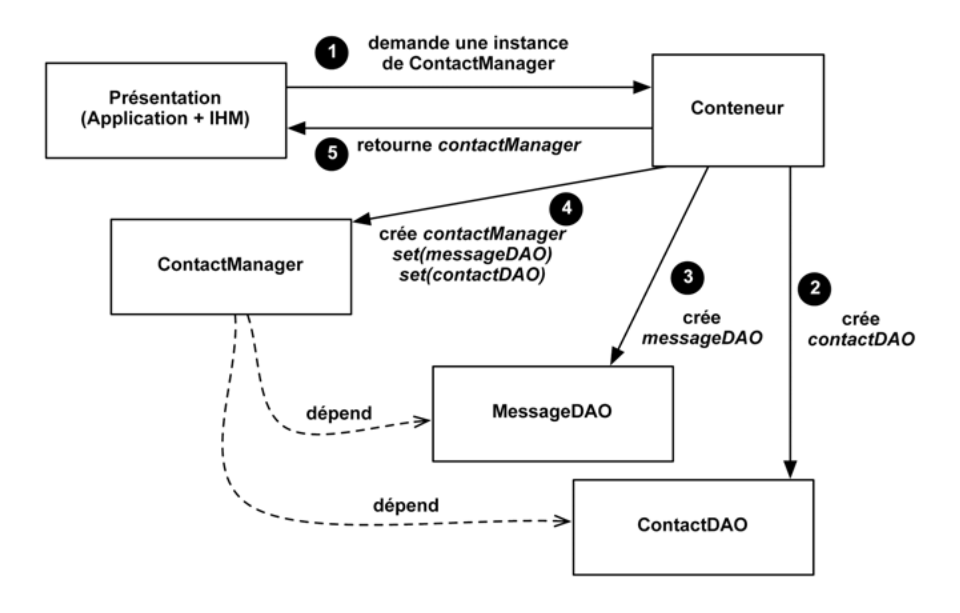


Figure 2.7 — Gestion des dépendances avec IoC

Pour reprendre l'exemple précédemment utilisé, le conteneur IoC gérant les dépendances entre les objets, le développeur n'a plus qu'à préciser celles-ci au conteneur (*via* un fichier de configuration XML ou par le biais des annotations). C'est l'application qui fera appel au conteneur afin de récupérer les objets dont elle a besoin (fig. 2.7).

> **Remarque :** on parle souvent d'injection de dépendances (*dependency injection*) pour préciser l'utilisation du *design pattern* IoC. C'est cependant un abus de langage, l'IoC regroupant plusieurs aspects, dont l'injection de dépendances.

En résumé

Très longtemps décrié, le système EJB est aujourd'hui remis à neuf. Fort de son utilisation des nouveautés même de J2SE 5.0, c'est principalement grâce à la communauté et à ses retours d'expériences que cette « révolution » a pu avoir lieu. Bien que simplifiée, l'utilisation des EJB n'en est pas moins complexe et demande une bonne connaissance de son fonctionnement pour effectuer les choix adéquats.

3

Les *Session Beans*

Objectif

Les principes fondamentaux de l'architecture métier définissent la création de services en tant qu'intermédiaires entre les applications clientes et l'accès aux données. Au sein d'une architecture Java EE, ce sont des EJB qui remplieront cette fonction : les *Session Beans*. Plus que de simples classes composées de propriétés et de méthodes, ces *Session Beans* sont de véritables passerelles de services au sein même de l'application, permettant à tout type de client de les interroger.

Après une présentation générale du concept des *Session Beans*, nous exposerons, de manière pratique, la création de ceux-ci avec EJB 2, puis avec EJB 3. Nous finirons par une rapide présentation de la mise en place de services web grâce aux *Session Beans*.

3.1 RÔLE DES SESSION BEANS

3.1.1 Qu'est-ce qu'un Session Bean ?

Un Session Bean est une application côté serveur permettant de fournir un ou plusieurs services à différentes applications clientes. Un service sert, par exemple, à récupérer la liste des produits d'une boutique, à enregistrer une réservation ou encore à vérifier la validité d'un stock. Il peut également représenter un *workflow*[1]. Un Session Bean peut être considéré comme un *storyboard* dans le monde de l'animation cinématographique : il définit les étapes successives afin d'aboutir à un objectif final.

1. *Workflow* : gestion électronique de processus métier. Un *workflow* décrit les étapes et tâches à effectuer automatiquement tout au long d'un processus opérationnel. Par exemple, c'est la gestion des états d'un article (idée, création, écriture, relecture, validation) dans le cas d'une chaîne éditoriale.

Les Session Beans constituent donc les briques de la logique métier d'une application. Ils traitent les données et effectuent les opérations liées à la logique de l'entreprise.

Les Session Beans font office de « pont » entre les clients et les données. Alors que les Entity Beans (expliqués dans le chapitre suivant) servent à accéder aux données (ajout, modification, suppression...), les Session Beans offrent généralement un accès en lecture seule sur celles-ci. Ils sont divisés en deux types : « Stateless » et « Stateful ».

3.1.2 Les Session Beans Stateless et Stateful

Le choix du type *Stateless*[1] ou *Stateful*[2] s'appuie sur l'interaction désirée entre le client et le Session Bean.

Un Stateless Session Bean est une collection de services dont chacun est représenté par une méthode. *Stateless* signifie que le service est autonome dans son exécution et donc qu'il ne dépend pas d'un contexte particulier ou d'un autre service. Le point important réside dans le fait qu'aucun état n'est conservé entre deux invocations de méthodes. Lorsqu'une application cliente appelle une méthode d'un Session Bean, celui-ci exécute la méthode et retourne le résultat. L'exécution ne se préoccupe pas de ce qui a pu être fait avant ou ce qui pourra être fait après. Ce type d'exécution est typiquement le même que celui du protocole HTTP (mode déconnecté).

Les Stateless Session Beans tendent à être généraux afin de pouvoir être réutilisés dans d'autres contextes.

L'avantage du type *Stateless* est sa performance. En effet, plusieurs clients utilisent la même instance de l'EJB (fig. 3.1).

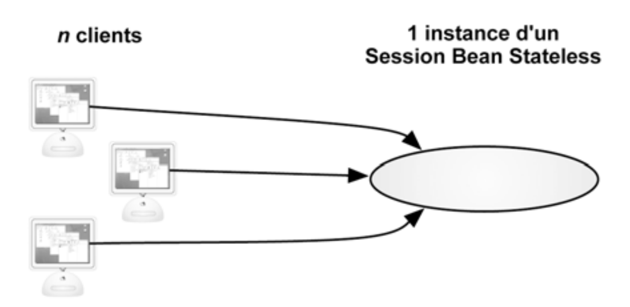


Figure 3.1 — Liaison entre clients et Session Bean Stateless

Un Stateful Session Bean est une extension de l'application cliente. Il introduit le concept de session entre le client et le serveur. On parle précisément d'état conversationnel pour qualifier ce type de communication. De ce fait, une méthode

1. *Stateless* : signifie littéralement « sans état ».
2. *Stateful* : signifie littéralement « avec état ».

appelée sur l'EJB peut lire ou modifier les informations sur l'état conversationnel. Cet EJB est partagé par toutes les méthodes pour un unique client. Contrairement au type *Stateless*, les Stateful Session Beans tendent à être spécifiques à l'application. Le caddie virtuel est l'exemple le plus commun pour illustrer l'utilisation d'un Stateful Session Bean.

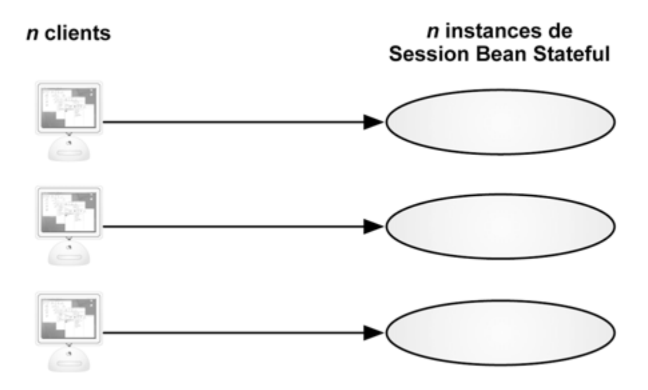


Figure 3.2 — Liaison entre clients et Session Bean Stateful

Dans le cas d'un *Stateful*, chaque client est lié à une instance de l'EJB (fig. 3.2). Ce type de Session Bean consomme donc davantage de mémoire que le type *Stateless*. De plus, le travail et le maintien d'association constituent une tâche supplémentaire importante pour le conteneur. Il en résulte une moins bonne montée en charge et parfois une dégradation des performances lorsqu'une application utilise le type *Stateful* abusivement et sans raison.

3.1.3 Quand utiliser les Session Beans ?

Voici quelques conditions concernant l'utilisation des *Session Beans* dans un système d'entreprise :

- À n'importe quel instant, seulement un client a accès à l'instance du Session Bean.
- L'état du Bean n'est pas persistant, il n'existe que pour une courte durée (environ quelques heures).
- Le service peut être accessible également *via* un service web.

Les Stateful Session Beans sont appropriés si l'une des conditions suivantes (non exhaustive) est vraie :

- L'état du Bean représente l'interaction entre le Bean et un client particulier.
- Le Bean doit conserver les informations concernant le client durant l'exécution des méthodes.

- Le Bean fait la liaison entre le client et d'autres composants de l'application, présentant une vue simplifiée au client.
- En coulisse, le Bean contrôle le workflow de plusieurs Entreprise Beans (c'est donc une façade).

Pour améliorer les performances, vous pouvez choisir d'utiliser un Stateless Session Bean s'il a l'une de ces caractéristiques :

- L'état du Bean n'a pas de donnée spécifique à un client.
- Dans un seul appel de méthodes, le Bean accomplit une tâche générique pour tous les clients. Par exemple, envoyer un email qui confirme un ordre en ligne.
- Le Bean récupère d'une base de données un ensemble de données en lecture seule qui sont souvent utilisées par les clients. Un tel Bean, par exemple, pourrait récupérer les lignes d'une table qui représentent les produits qui sont en vente ce mois.

Le choix du type de Session Bean n'est pas toujours évident. Cependant, dans une grande majorité des cas, les services offerts sont « statiques », c'est-à-dire indépendants du client qui les appelle. Dans de tels cas, préférez le type Stateless par défaut.

3.2 EJB 2 : ÉCRITURE D'UN SESSION BEAN

La spécification EJB 2 impose le développement de plusieurs éléments : la classe du Bean, les interfaces et la déclaration du Bean dans le descripteur de déploiement.

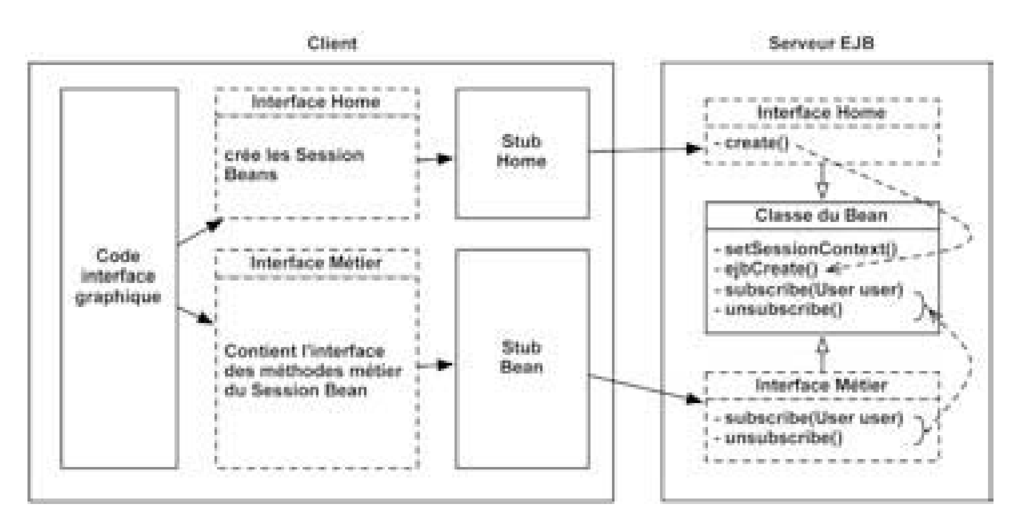


Figure 3.3 — Anatomie d'un Session Bean

Les différentes liaisons entre les composants (fig. 3.3) sont expliquées tout au long de ce chapitre.

3.2.1 La classe du Bean

La classe du Bean permet de définir le code métier des services de l'application. Le point sur lequel le développeur doit se focaliser est donc l'écriture de ses méthodes. L'exemple utilisé dans cette partie est le Stateful Session Bean `RichClientService`.

```
public class RichClientServiceBean implements SessionBean {

  // variables d'instance liées au cycle de vie de l'EJB Stateful
  private User currentUser;

  private SessionContext sessionContext;

  /**
   * Méthode de création de l'EJB
   */
  public void ejbCreate() throws CreateException {}

  /**
   * Méthodes du cycle de vie
   */

  public void setSessionContext(SessionContext sessionContext)
                        throws EJBException, RemoteException {
    this.sessionContext = sessionContext;
  }

  public void ejbRemove() throws EJBException, RemoteException { }

  public void ejbActivate() throws EJBException, RemoteException { }

  public void ejbPassivate() throws EJBException, RemoteException { }


  // méthodes protégées disponibles uniquement pour l'instance de l'EJB
  protected CommonServiceLocal getCommonService() throws Exception {
    CommonServiceLocalHome home = (CommonServiceLocalHome) ServiceLocator
      .getInstance().getLocalHome("java:comp/env/ejb/CommonService");
    return home.create();
  }

  protected TransactionLocalHome getTransactionHome() throws Exception {
    return (TransactionLocalHome)
ServiceLocator.getInstance().getLocalHome("java:comp/env/ejb/Transaction");
  }

  protected AccountInfoLocalHome getAccountInfoHome() throws Exception {
    return (AccountInfoLocalHome)
ServiceLocator.getInstance().getLocalHome("java:comp/env/ejb/AccountInfo");
  }

  /**
   * Méthodes métiers
   */

  /**
   * Retourne l'utilisateur lié à l'instance de ce bean
   */
  public User getCurrentUser() {
```

```
    return currentUser;
  }

  /**
   * Méthode d'inscription d'un utilisateur
   */
  public User subscribe(User user) throws ValidationException {
    ValidationException validationException = new ValidationException();

    try {

      // vérification du nom
      if (user.getLastName() == null) {
        validationException.addError(new ValidatorError(
          "Le champ est requis", "lastName"));
      }
      // vérification du prénom
      if (user.getFirstName() == null) {
        validationException.addError(new ValidatorError(
          "Le champ est requis", "firstName"));
      }
      // vérification de l'adresse
      if (user.getAddress() == null) {
        validationException.addError(new ValidatorError(
          "Le champ est requis", "address"));
      }
      // vérification du compte
      if (user.getAccountInfo() == null) {
        validationException.addError(new ValidatorError(
          "Le champ est requis", "accountinfo"));
      }

      // si des erreurs se sont produites pendant la validation
      // on lève une exception
      if (validationException.getErrors().size() > 0) {
        throw validationException;
      }

      // ajoute l'utilisateur
      currentUser = getCommonService().addUser(user);

      // retourne l'utilisateur
      return currentUser;
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }

  /**
   * Méthode de désinscription de l'utilisateur
   */
  public void unSuscribe() {
    CommonServiceLocal commonServiceLocal;
    try {
      commonServiceLocal = getCommonService();
      commonServiceLocal.removeUser(currentUser.getId());
      currentUser = null;
    } catch (Exception e) {
```

```
            throw new RuntimeException(e);
        }
    }
    ...
}
```

La classe du Session Bean `RichClientService`, doit commencer par implémenter l'interface `javax.ejb.SessionBean`. Celle-ci assure au conteneur que la classe est bien celle d'un Session Bean et qu'elle implémente correctement les méthodes du cycle de vie.

Une fois ces méthodes implémentées, il faut définir une méthode `ejbCreate()`. Celle-ci est appelée par le conteneur lorsqu'un client tente de récupérer une instance de l'EJB à partir de l'interface *home*. Seul un Stateful Session Bean peut définir plusieurs surcharges de `ejbCreate()`. Les initialisations liées à l'EJB peuvent être faites à l'exécution de cette méthode.

Les méthodes `suscribe()` ou `unSuscribe()` de l'exemple précédent sont des méthodes métiers permettant d'inscrire un utilisateur et de désinscrire l'utilisateur lié au Stateful Session Bean courant.

Méthodes du cycle de vie

Les méthodes du cycle de vie permettent au développeur d'effectuer des opérations lorsque l'état de l'instance du Bean est modifié par le conteneur. Les différentes méthodes de gestion du cycle de vie pour un Session Bean sont :

- `ejbActivate()`
- `ejbPassivate()`
- `ejbRemove()`
- `ejbCreate()`
- `setSessionContext(javax.ejb.SessionContext)`

Ces méthodes sont appelées automatiquement par le conteneur à des moments précis (fig. 3.4).

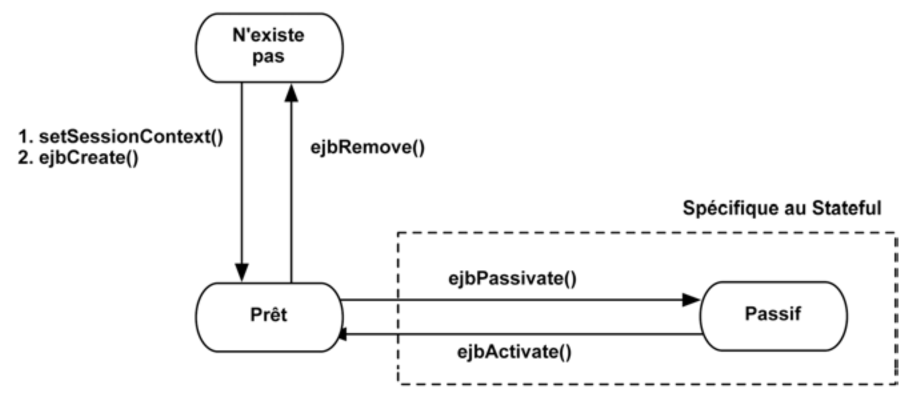


Figure 3.4 — Cycle de vie d'un Session Bean

La méthode `setSessionContext()` est appelée après la création de l'instance du Session Bean. Elle permet au développeur de récupérer un objet `SessionContext`, passé en argument. Cet objet permet alors d'accéder au contexte d'exécution du Session Bean. Concrètement, le développeur se sert de cet objet pour récupérer des références vers l'instance courante de l'EJB *via* `getEJBLocalObject()` ou `getEJBObject()` respectivement pour l'accès local ou distant. Le développeur utilise généralement ces méthodes lorsqu'il veut passer une référence de l'EJB courant (lui-même) en tant qu'argument d'une méthode ou en résultat.

> **Remarque :** les méthodes `getEJBLocalObject()` et `getEJBObject()` doivent être utilisées à la place de l'opérateur `this` qui n'est pas représentatif de l'instance de l'EJB.

Les méthodes `ejbActivate()` et `ejbPassivate()` sont respectivement appelées lors de la sauvegarde et la restauration du Session Bean. On entend par sauvegarde, la sérialisation de l'instance de celui-ci sur un disque dur, par exemple. À l'inverse, la restauration permet de charger l'instance du Session Bean en mémoire. Ces principes d'activation et de passivation offrent alors la possibilité au conteneur de réduire la mémoire utilisée et donc d'accroître les performances. Ces méthodes sont généralement utilisées dans le cadre des Stateful Session Beans, lorsqu'un client est inactif.

La méthode `ejbRemove()` est appelée au moment de la suppression de l'instance par le conteneur. Elle permet au développeur d'effectuer les opérations de nettoyage (déconnexion à une base de données...) à la suppression de l'instance.

3.2.2 Les interfaces

Les interfaces d'un EJB permettent au client d'avoir connaissance des méthodes qu'il a à sa disposition, sans être dépendant de la classe de celles-ci. Avec EJB 2, en plus de définir le code de l'EJB, il est nécessaire de réaliser au moins deux interfaces. La première définit les méthodes de création de l'EJB, c'est l'interface de fabrication *home*. La seconde regroupe les méthodes métiers disponibles pour le client, c'est l'interface métier *business*.

Il existe, cependant, plusieurs types de clients : les locaux (*local*) et les distants (*remote*). Ces deux types d'interfaces, *home* et *business*, se déclinent donc, elles aussi, pour ces deux types d'accès. Il existera donc quatre interfaces distinctes :

- *local home*
- *home* (pour l'accès « *remote home* »)
- *local* (pour l'accès « *local business* »)
- *remote* (pour l'accès « *remote business* »)

> **Attention :** ne vous perdez pas dans les noms des interfaces ! Remarquez l'appellation de certaines interfaces volontairement tronquée. Ces noms se basent sur la nomenclature de déclaration interfaces dans le descripteur de déploiement, que nous étudions dans les parties suivantes.

Différences entre « local » et « remote »

L'interface *remote* est destinée aux clients distants. Tous les appels de méthodes sont donc susceptibles de remonter des exceptions de type `java.rmi.RemoteException` qui pourraient être levées suite à une erreur de connectivité entre le client et le conteneur d'EJB. En effet, un client distant ne s'exécute généralement pas dans la même machine virtuelle que celle du serveur.

L'interface *local* est, quant à elle, destinée aux clients locaux, qui s'exécutent dans la même machine virtuelle que le serveur (typiquement, d'autres EJB ou une servlet s'exécutant sur le même serveur). Celle-ci permet également de mettre à disposition du client les méthodes métiers de l'EJB. À la différence de l'interface distante, ses méthodes n'ont pas à remontrer d'exception, RMI n'intervenant pas dans le processus de communication avec un client local.

Interface « home »

L'interface *home* est utilisée par le client pour récupérer une instance d'un EJB. En l'occurrence, elle contient la ou les méthodes `create()` qui permettent aux clients de récupérer une instance de l'EJB. Chaque EJB a sa propre interface *home* qui doit hériter de `javax.ejb.EJBHome` ou de `javax.ejb.EJBLocalHome`. Celles-ci définissent respectivement si le type d'accès est distant ou local.

```
public interface RichClientServiceHome extends javax.ejb.EJBHome {
  public RichClientService create() throws
javax.ejb.CreateException,java.rmi.RemoteException;
}
```

Cette interface hérite bien de `javax.ejb.EJBHome` et déclare la méthode `create()` permettant au client de récupérer une instance implémentant l'interface métier `RichClientService`. Cette méthode peut lever des exceptions de type `RemoteException` car elle est accessible à distance.

Interface « business »

L'interface *business* (ou métier), comme son nom l'indique, regroupe l'ensemble des méthodes métiers disponibles pour le client. Chaque méthode déclarée dans cette interface doit avoir son implémentation dans la classe de l'EJB. Dans le cas contraire, des erreurs seront lancées au moment du déploiement. Une interface métier doit obligatoirement hériter de `javax.ejb.EJBObject` ou de `javax.ejb.EJBLocalObject`. Celles-ci définissent respectivement si le type d'accès est distant ou local.

```
public interface RichClientService extends javax.ejb.EJBObject {
  public com.labosun.stockmanager.entity.User getCurrentUser(  ) throws
java.rmi.RemoteException;

  public com.labosun.stockmanager.entity.User subscribe(
com.labosun.stockmanager.entity.User user) throws
com.labosun.stockmanager.exception.ValidationException,
java.rmi.RemoteException;
```

```
    public void unSuscribe() throws java.rmi.RemoteException;
    // ...autres méthodes métiers
}
```

L'interface `RichClientService` hérite bien de `javax.ejb.EJBObject` et l'ensemble des méthodes déclarées existent dans la classe `RichClientServiceBean` présentée précédemment. Toutefois, chaque méthode de l'interface déclare la possibilité de lever des `java.rmi.RemoteException` car l'accès à ces méthodes se fait à distance.

Dans le cas d'une interface métier *local*, les méthodes ne déclarent pas cette possibilité de levée d'exception, RMI n'étant pas utilisé entre le client et le serveur.

3.2.3 Le descripteur de déploiement

Après avoir réalisé les classes des Beans et les interfaces correspondantes, il faut déclarer le Session Bean dans le descripteur de déploiement EJB « ejb-jar.xml », comme le montre le code suivant. Celui-ci est lu par le conteneur lors du démarrage de l'application. Le conteneur est alors en mesure d'initialiser l'ensemble des EJB déclarés ainsi que les services associés.

```
<session >
   <description><![CDATA[EJB utilisé par le client riche]]></description>
   <ejb-name>RichClientService</ejb-name>

  <home>
    com.labosun.stockmanager.ejb2.session.interfaces.RichClientServiceHome
  </home>
  <remote>
    com.labosun.stockmanager.ejb2.session.interfaces.RichClientService
  </remote>
  <ejb-class>
    com.labosun.stockmanager.ejb2.session.ejb.RichClientServiceBean
  </ejb-class>
  <session-type>Stateful</session-type>
  <transaction-type>Container</transaction-type>
</session>
```

La balise `<session>` permet de déclarer un Session Bean. Les balises enfants `<home>`, `<remote>`, `<ejb-class>`, `<transaction-type>` définissent respectivement l'interface *home*, l'interface *remote*, la classe du Bean et le type de transaction associés.

La déclaration des interfaces locales (*local home* et *local business*), si il y a, se fait grâce aux balises `<local-home>` et `<local>`. La balise `<session-type>` définit le type du Session Bean, elle accepte les valeurs `Stateful` ou `Stateless`.

3.3 EJB 3 : ÉCRITURE D'UN SESSION BEAN

L'API a été grandement simplifiée en EJB 3. Dorénavant, il n'est plus nécessaire de maintenir différentes interfaces suivant le type d'accès que l'on souhaite donner au

client. Le développeur peut désormais se concentrer sur l'aspect métier de l'application sans se soucier des contraintes de « tuyauterie » liées à la technologie EJB 2.

Ainsi, le développeur n'a plus à créer les interfaces *home* et peut se concentrer sur l'interface *business*. Nous verrons également, que le couplage avec les interfaces de la spécification n'est plus nécessaire.

Les éléments à mettre en place pour un Session Bean sont la classe du Bean et l'interface *business* (métier). Désormais, une seule interface est nécessaire, et la déclaration au sein du descripteur de déploiement n'est plus obligatoire.

3.3.1 La classe du Bean

La classe d'un Session Bean EJB 3 n'a pas à implémenter d'interface, comme c'est le cas avec un Session Bean EJB 2. Il suffit ici d'utiliser les annotations `@Remote` ou `@Local` pour définir respectivement si le Session Bean fournit les méthodes à des clients distants ou locaux. Il en est de même pour les types de Session Beans, grâce aux annotations `@Stateless` ou `@Stateful`.

```
@Stateful
@Remote( { RichClientService.class })
public class RichClientServiceBean implements RichClientService {

...
  private User currentUser;

  public User getCurrentUser() {
    return commonService.findUser(currentUser.getId());
  }

  public User subscribe(User user) throws ValidationException {
    ValidationException validationException = new ValidationException();

    // vérification du nom
    if(user.getLastName() == null) {
      validationException.addError(new ValidatorError(
        "Le champ est requis", "lastName"));
    }
    // vérification du prénom
    if (user.getFirstName() == null) {
      validationException.addError(new ValidatorError(
        "Le champ est requis", "firstName"));
    }
    // vérification de l'adresse
    if (user.getAddress() == null) {
      validationException.addError(new ValidatorError(
        "Le champ est requis", "address"));
    }
    // vérification du compte
    if (user.getAccountInfo() == null) {
      validationException.addError(new ValidatorError(
        "Le champ est requis", "accountinfo"));
  }
```

```
      // si des erreurs se sont produites pendant la validation
      // on lève une exception
      if (validationException.getErrors().size() > 0) {
        throw validationException;
      }

      // inscrit l'utilisateur
      User newUser = commonService.addUser(user);

      // sauvegarde l'entité
      currentUser = newUser;
      return newUser;
    }
    public void unSuscribe() {
      commonService.removeUser(getCurrentUser());
      currentUser = null;
    }
    ...
  }
```

Dans cet exemple, la classe est annotée avec `@Stateful` afin de la déclarer en tant que Stateful Session Bean. L'annotation `@Remote` permet de spécifier les interfaces distantes utilisables par les clients.

Contrairement à un Session Bean EJB 2, la classe du Session Bean EJB 3 implémente directement la ou les interfaces métiers utilisées. Cela élimine toutes les erreurs d'inattention que les développeurs pouvaient faire (oubli d'implémentation de telle méthode métier définie dans l'interface...). De plus, cette nouvelle façon de faire semble beaucoup plus intuitive que la précédente. Nous comprenons plus facilement la liaison avec l'interface métier.

Autre différence, la classe ne contient plus directement l'implémentation des méthodes du cycle de vie (`setSessionContext()`, `ejbActivate...`). Néanmoins, il est toujours possible de spécifier des méthodes qui seront appelées lors du changement d'état dans le cycle de vie.

3.3.2 Les méthodes du cycle de vie

Afin de gérer la vie d'un Session Bean, le conteneur EJB doit procéder à différentes étapes, qui constituent le cycle de vie de celui-ci.

Tout d'abord, pour exploiter un Session Bean, celui-ci doit être instancié. De façon très simplifiée, le conteneur s'occupe de cette instanciation par l'appel de la méthode `newInstance()`. Celle-ci est disponible à partir de l'objet `Class` lié à la classe du Session Bean, comme le montre cet exemple :

```
RichClientServiceBean.class.newInstance();
```

Attention : ceci implique donc que la classe d'implémentation du Session Bean dispose d'un constructeur public, sans argument.

Le conteneur va ensuite analyser cette instance afin de déceler les éventuelles injections de dépendance à effectuer. L'injection de dépendance est le processus qui permet au conteneur d'initialiser des propriétés de l'EJB automatiquement. Le développeur spécifie ces dépendances grâce à des annotations telles que `@EJB`, `@Resource`… que nous verrons par la suite en détail.

Après cela, le conteneur va éventuellement exécuter les méthodes *callback interceptors* annotées avec `@PostConstruct`.

> **Remarque :** notez que les annotations en question ne sont pas strictement liées aux spécifications EJB 3 : elles font partie d'une spécification beaucoup plus généraliste, la JSR-250 Common Annotations for the Java Platform v1.0, elle-même intégrée à la plate-forme Java EE 5 (JSR-244 Java Platform, Enterprise Edition 5).

Les méthodes *callback interceptors* doivent être annotées avec les annotations de *callback*.

```
@Stateful
public class ShoppingCartBean implements ShoppingCart {
  private float total;
  private List productCodes;
  public int someShoppingMethod(){...};

   ...

  @PostConstruct
  public void beginShopping() {...};

  @PreDestroy
  public void endShopping() {...};

  ...

}
```

Les méthodes `beginShopping()` et `endShopping()` sont appelées respectivement, par le conteneur, après la création de l'instance du Bean et juste avant sa suppression.

Il en est de même pour toutes les autres phases du cycle de vie d'un Session Bean. Selon l'état du Bean et l'état que doit prendre celui-ci, le conteneur appellera l'une ou l'autre des méthodes *callback interceptors* (fig. 3.5 et 3.6).

Les Session Beans supportent les *callback interceptors* suivants :

- `@PostConstruct` qui intervient après toutes les dépendances d'injection effectuées par le conteneur et avant le premier appel de la méthode métier.
- `@PreDestroy` qui est appelé au moment où l'instance du Bean est détruite (lors de la suppression du Bean ou à l'arrêt de l'application).
- `@PrePassivate` (Stateful uniquement) qui permet de spécifier une méthode qui sera appelée par le conteneur EJB lorsque le Bean a été inactif et qu'il a été jugé nécessaire de passer celui-ci dans un état sérialisable. La méthode en

question doit s'assurer que les ressources maintenues en instance sont libérées ou sérialisables à leur tour. On parle de « passivation ». L'intérêt de ce principe est de libérer la mémoire employée par le Stateful inutilisé pour le moment. Le conteneur peut alors sérialiser l'instance sur un support externe (disque dur…).

- `@PostActivate` (Stateful uniquement) qui joue le rôle inverse de `@PrePassivate`. Cette annotation permet de spécifier la méthode qui sera appelée par le conteneur lorsqu'un Bean devra être réactivé de son état de passivation. Le Bean devra retrouver un état opérationnel en récupérant les ressources éventuellement libérées lors de la « passivation ». On parle d'« activation ».

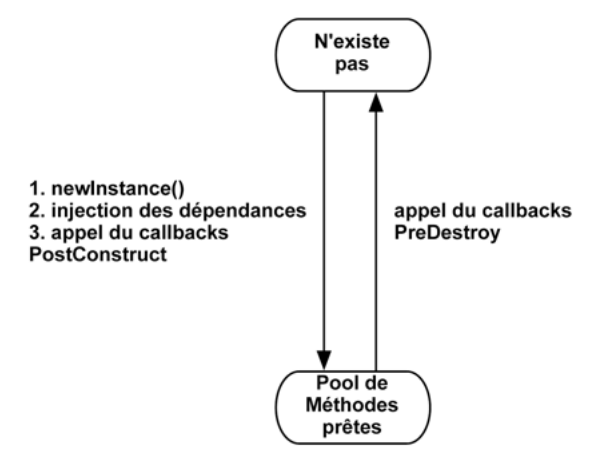


Figure 3.5 — Cycle de vie d'un Stateless Session Bean (EJB 3)

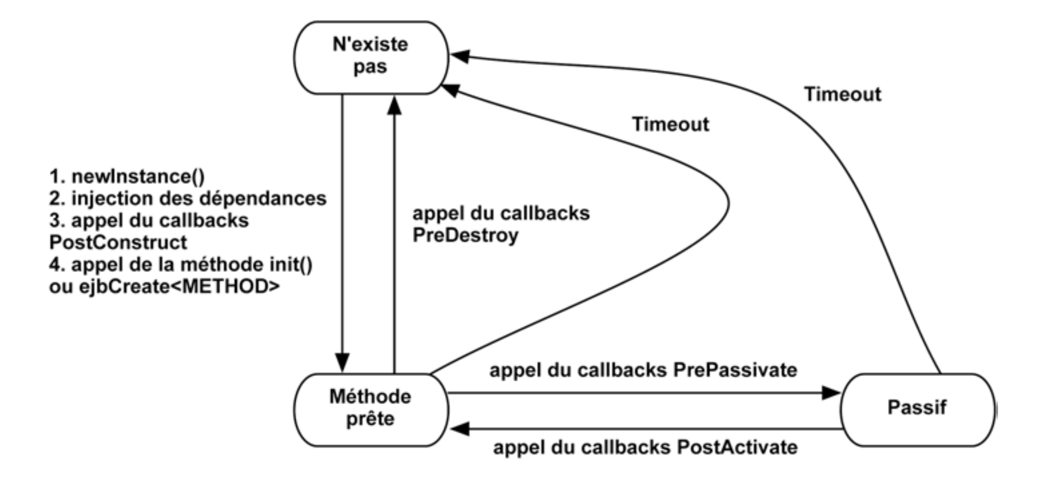


Figure 3.6 — Cycle de vie d'un Stateful Session Bean (EJB 3)

L'état « passivé » est réservé aux Stateful Session Beans car ce sont les seuls à partager un état avec le client. Le cycle de vie pour ce type de composant commence au premier appel d'une méthode métier. Le conteneur crée alors une instance *via* la méthode `Class.newInstance()`. Il injecte alors les dépendances et appelle les méthodes annotées avec `@PostConstruct`. La grande différence avec le type Stateless est que le Stateful intègre le concept de « passivation ». L'implémentation de ce concept est spécifique au fournisseur de conteneur et peut donc varier d'un serveur à l'autre. Dans tous les cas, le conteneur appelle les méthodes annotées avec `@PrePassivate` et `@PostActivate` au moment de la passivation et de l'activation du composant.

L'instance est supprimée lorsque le client appelle une méthode annotée avec `@Remove` ou lorsque le temps imparti pour la session est dépassé (*timeout*).

Il existe deux possibilités de déclarer ces méthodes *callback interceptors* :

- La première solution est, comme nous venons de le voir, de définir des méthodes callback directement dans la classe du composant. Dans ce cas, la signature des méthodes doit respecter la syntaxe suivante, où <METHOD> correspond à un nom de méthode choisi :

```
public void <METHOD> ()
```

- La seconde solution consiste en la séparation des méthodes de gestion du cycle de vie dans des classes intercepteurs (voir paragraphe 3.3.5).

Remarque : notez qu'il est toujours possible de conserver l'usage des méthodes `ejbRemove()`, `ejbActivate()`, `ejbPassivate()` (interface `javax.ejb.SessionBean`) et `ejbCreate` comme c'était le cas en EJB 2. Elles seront traitées comme si elles étaient annotées par `@PreDestroy`, `@PostActive`, `@PrePassivate` et `@PostConstruct`. L'utilisation de l'interface `javax.ejb.SessionBean` limitera l'utilisation de ces annotations à ces seules méthodes. Il convient donc de ne pas utiliser cette interface si la flexibilité des annotations est privilégiée.

3.3.3 Particularités du Stateful Session Bean

Comme nous l'avons détaillé au début de cette partie, chaque instance d'un Stateful Session Bean est associée à un client unique. Ce type de composant maintient l'état conversationnel avec le client. Les variables d'instances de ce composant sont alors liées au client et leurs valeurs conservées d'un appel de méthodes à un autre.

Toutefois, cela ne signifie en aucun cas qu'un Stateful Session Bean est persistant. Le serveur d'applications peut utiliser un système de tampon (*swap*) pour des soucis d'optimisation de mémoire (concept de passivation)

Un Stateful Session Bean sert habituellement à gérer un processus s'effectuant en plusieurs étapes (achats sur un site marchant, réserver un voyage, envoyer une newsletter…).

Prenons l'exemple d'un site de commerce *online*. Le panier peut être modélisé grâce à un Stateful Session Bean. Une vente est réalisée après plusieurs étapes :

- Choix des articles
- Création d'un compte client ou utilisation d'un compte existant
- Choix du paiement de la commande
- Paiement
- Enregistrement de la commande

Il est impératif de sauvegarder l'état de la conversation entre ces différentes étapes afin de ne pas perdre d'informations.

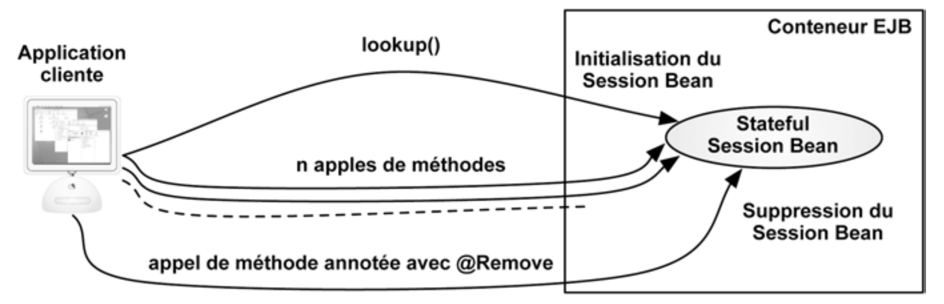


Figure 3.7 — Création et suppression d'un Stateful Session Bean

La vie d'un Session Bean Stateful démarre lors de la récupération de celui-ci par l'application cliente (*via* l'injection ou JNDI). Elle se termine si le temps imparti à la durée de vie est dépassé ou si le client appelle une méthode annotée avec `@Remove`.

> **Attention :** à chaque appel de la méthode `Context.lookup()` une instance du Stateful Session Bean est créée. Veillez à bien gérer la création et la suppression de ces instances pour ne pas surcharger la mémoire.

3.3.4 Les interfaces métiers

Avec EJB 3, les interfaces métiers sont dites des « pures ». C'est-à-dire qu'elles n'ont plus besoin d'hériter de `javax.ejb.EJBObject` ou de `javax.ejb.EJBLocalObject`. Les méthodes de cette interface doivent cependant répondre aux contraintes suivantes :

- Leur nom est libre tant qu'il ne commence pas par « `ejb` » pour éviter tout conflit avec les méthodes *callback interceptors*.
- Elles doivent être déclarées comme étant `public`.
- Elles ne doivent pas être déclarées comme `final` ou `static`.
- Dans le cas d'un accès de type *remote*, le type de retour et le type des arguments doivent être compatibles avec des transports de type RMI/IIOP.

- De même, dans le cas d'un service web, le type de retour et des arguments doivent être compatibles avec un transport de type JAX-WS/JAX-RPC.

Les méthodes des interfaces métiers peuvent désormais lever des exceptions liées à l'application, via la clause `throws`. Il n'est, de même, plus nécessaire de spécifier, pour les méthodes de l'interface distante, l'exception `java.rmi.RemoteException`.

Si un problème survient au niveau du protocole, une exception de type `javax.ejb.EJBException` est levée par le conteneur. Celle-ci étant de type « runtime », le développeur n'a pas à la spécifier dans l'élaboration de ses interfaces.

Voici un exemple d'interface métier de type *local* :

```
@Local
public interface CommonService {

  public void removePortfolio(com.labosun.stockmanager.entity.Portfolio
portfolio);

  public void savePortfolio(com.labosun.stockmanager.entity.Portfolio
portfolio);

  // autres méthodes métiers
...
}
```

Nous définissons ici simplement l'interface qui contient les méthodes métiers accessibles par une application cliente. Nous précisons, ici, que l'accès à ces méthodes devra être fait de manière *local* avec l'annotation `@Local`. Bien que possible dans l'interface, cette précision est généralement faite au niveau de la classe du Session Bean. Cela permet une entière compatibilité avec des clients utilisant la JVM 1.4 qui ne gère pas les annotations.

À l'opposé, l'annotation `@Remote` définit les interfaces métiers accessibles par les applications clientes distantes.

3.3.5 Les intercepteurs : gestion avancée du cycle de vie

L'utilisation de méthodes de type *callback interceptors* est une pratique facile à mettre en place. Toutefois, l'inconvénient principal de cette approche est le « mélange » qu'elle génère entre les méthodes métiers et celle du cycle de vie des Session Beans. Ce rassemblement peut porter à confusion si la gestion du cycle de vie est importante.

Il est cependant possible de définir cette gestion du cycle de vie dans des classes dédiées, appelées classe *interceptor*. Leur écriture est soumise aux mêmes restrictions que l'écriture des Session Beans ; elles doivent donc, par exemple, posséder un constructeur public sans argument.

L'association d'une classe d'intercepteur à un Session Bean se fait à l'aide de l'annotation `@Interceptors` qui prend en argument la ou les classes d'intercepteur(s). Si plusieurs classes sont spécifiées, l'ordre d'énumération sera celui utilisé lors des différentes « interceptions » du cycle de vie.

Le cycle de vie d'un intercepteur est lié à l'EJB auquel il est associé. En effet, une instance est créée lors de l'instanciation du Session Bean et elle est détruite lors de la destruction de celui-ci.

Un intercepteur peut servir à intercepter les changements du cycle de vie du Session Bean, mais peut aussi servir à intercepter les appels aux méthodes métier de celui-ci.

Dans le cas de la gestion du cycle de vie (`@PostConstruct`, `@PreDestroy`...), il est possible d'annoter, au sein de la classe d'intercepteurs, une méthode ayant la signature `public void <METHOD>(InvocationContext ctx)` avec le type de *callback* à gérer. Voici un exemple de classe d'*interceptor* :

```
@Interceptors({MyInterceptor.class})
@Stateless
public class MyEnterpriseBean {
  ...
}
```

```
public class MyInterceptor {
  @PostConstruct
  public void myMethod(InvocationContext ctx) throws Exception {
    System.out.println("Before PostConstruct event");
    try {
      ctx.proceed();
    } finally {
      System.out.println("After PostConstruct event");
    }
  }
  ...
}
```

Dans le cas de l'interception des méthodes métier, l'annotation utilisée est `AroundInvoke`, avec une signature du type `Object <METHOD>(InvocationContext) throws Exception` :

```
public class MyInterceptor {
  @AroundInvoke
  public Object myBusinessInterceptor(InvocationContext ctx) throws Exception{
    System.out.println("Intercepting " + ctx.getMethod().getName());
    try {
      return ctx.proceed();
    } finally {
      System.out.println(ctx.getMethod().getName() + " done");
    }
  }
}
```

Ici, l'intercepteur s'appliquera à toutes les méthodes métier dont la classe est annotée *via* `@Interceptors(MyInterceptor.class)`. Néanmoins, il est également possible de ne spécifier que certaines méthodes, en annotant directement celles-ci.

Ce concept d'intercepteur est lié l'AOP[1]. L'AOP n'est pas le successeur de la POO[2] mais comble certaines limites de celle-ci. En effet en POO, les couches applicatives d'une application sont empilées. De ce fait, il est difficile d'élaborer une opération transversale à l'ensemble des couches. Ces opérations sont généralement liées à la gestion des traces (*logs*), de la sécurité, de la persistance (dans certains cas) ou encore de la performance (*benchmarks*). L'AOP résout en quelque sorte ce problème. Elle permet d'ajouter à un code existant, et de façon indépendante, des traitements transversaux par le biais d'intercepteur.

Attention : cette définition de l'AOP est plus que simplifiée. Nous avons voulu rester concis, ce concept n'étant pas l'objet principal de cet ouvrage.

Même si cette nouvelle méthode est une réelle évolution dans le monde du développement, elle reste très ardue pour les personnes non initiées. La chose importante à retenir est qu'elle permet d'appliquer un traitement transversal à un ensemble de méthodes et évite donc toute pollution de code non lié à la couche courante.

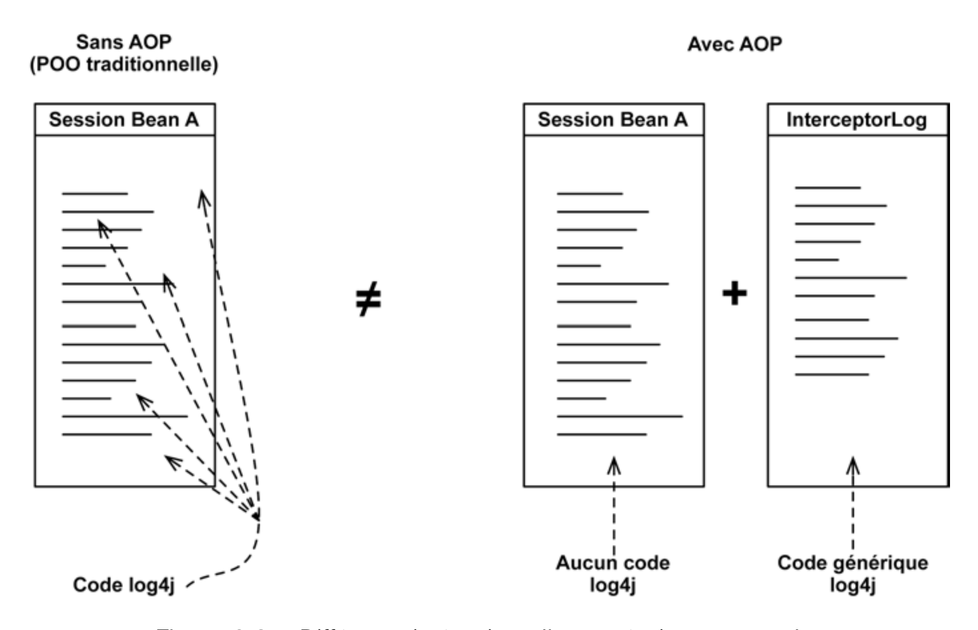


Figure 3.8 — Différentes intégrations d'une opération transversale

1. AOP (*Aspect-Oriented Programming*) : paradigme de programmation qui permet de réduire fortement les couplages entre les différentes couches d'une application (http://fr.wikipedia.org/wiki/Programmation_orientée_aspect).
2. POO (Programmation orientée objet).

La figure 3.8 schématise l'intégration d'un traitement type « journalisation » qui pourrait être fait avec l'API log4j. Dans le cas d'une implémentation 100 % POO, le développeur doit intégrer les appels liés à l'API dans le code métier du Session Bean. Alors que dans la seconde implémentation, le code du Session Bean est purement métier et une classe d'intercepteur séparée gère la journalisation. Il est à noter, cependant, que la précision de la journalisation dans le second cas est moins fine que dans le premier cas. En effet, l'intercepteur s'exécute au début de la méthode et à la fin de la méthode et non dans la méthode !

3.4 INTRODUCTION AUX SERVICES WEB

3.4.1 Qu'est-ce qu'un service web ?

Avant d'expliquer ce qu'est un service web (*Web Service*), nous allons voir pourquoi ils existent. Auparavant, pour mettre en place des applications distribuées, les développeurs devaient mettre en place des architectures type Corba dans le cas d'applications hétérogènes, RMI en environnement Java ou encore COM. Pour faciliter la coordination de différents systèmes hétérogènes, les grands éditeurs tels SUN, IBM, Oracle… ont décidé d'établir un standard de communication : les services web.

Un service web est un ensemble de protocoles et de normes. Ils sont utilisés pour échanger des données entre des applications hétérogènes. En effet, cela est rendu possible par l'utilisation d'un protocole de communication standard basé sur XML. Les standards utilisés sont :

- *Web Services Protocol Stack* : terme désignant la collection de standards utilisés par les services web.
- XML : les données échangées sont formatées en XML grâce à SOAP ou XML-RPC[1].
- WSDL (*Web Services Description Language*) : protocole de description de l'interface publique du service web. Cette interface regroupe l'ensemble des opérations disponibles et les prototypes (arguments typés) de celles-ci. Le format utilisé pour cette description est le XML
- UDDI (*Universal Description Discovery and Integration*) : protocole de recherche et d'enregistrement d'un service web.
  Même si cette technologie est encore jeune, elle offre de grandes possibilités et son utilisation risque d'exploser littéralement dans les architectures complexes (systèmes hétérogènes). On les utilise généralement dans les architectures type SOA[2].

1. XML-RPC (*XML Remote Procedure Call*) : cf. http://fr.wikipedia.org/wiki/XML-RPC.
2. SOA (*Service-Oriented Architecture*) : modèle d'interaction applicative mettant en œuvre des services (http://fr.wikipedia.org/wiki/Service_Oriented_Architecture).

3.4.2 Écriture d'un Web Service

Depuis Java EE 5, le développement de service web est extrêmement simple. Même s'il existe plusieurs façons d'en développer, nous allons vous présenter celle se basant sur les Stateless Session Beans.

Avec la spécification EJB 2, le développeur devait impérativement créer une interface désignant les méthodes accessibles *via* SOAP. Dans la spécification EJB 3, cela n'est plus imposé. Il est possible de créer un service web à partir d'un simple POJO et quelques annotations !

Voici un exemple simple de service web qui offre une passerelle SOAP sur des services d'un *Stateless Session Bean*.

```
@WebService
@Stateless
public class WebServiceAccessBean {

  @EJB
  private CommonService commonService;

  @WebMethod (operationName="loginUser")
  public String login(@WebParam(name="userName") String login,
                      @WebParam(name="password") String password) {
    ...
  }
}
```

La base de cet exemple n'est rien d'autre qu'une classe Java standard (POJO). Nous avons seulement ajouté l'annotation `@Stateless`, afin de la déclarer en tant que *Stateless Session Bean*. Les deux seuls éléments obligatoires pour définir un accès de type service web sont les annotations `@WebService` et `@WebMethod`. La première déclare le service web en lui-même, la seconde les méthodes disponibles par celui-ci. Voici les détails de cette première annotation :

- `name` : définit le nom du service web. Ce nom se trouve dans le fichier WSDL dans l'attribut `name` des balises `<portType>`. La valeur par défaut est le nom de la classe d'implémentation (`WebServiceAccessBean` dans l'exemple).
- `serviceName` : définit le nom du service. Ce nom se trouve dans le fichier WSDL dans l'attribut `name` des balises `<service>`. La valeur par défaut est le nom de la classe d'implémentation suivie du suffixe « `Service` » (`WebServiceAccessBeanService` pour l'exemple).
- `wsdlLocation` : définit l'URL, qu'elle soit relative ou absolue, d'un fichier WSDL existant. Par défaut, ce fichier est auto-généré lors du déploiement.
- `endpointInterface` : définit le nom complet de l'interface « endpoint » définissant les méthodes du service web. Cela permet au développeur de séparer le « contrat » (l'interface) de l'implémentation. L'implémentation de cette interface par le service n'est pas requise. L'intérêt de cette solution est également d'affecter l'ensemble des annotations de *mapping* Java vers le WSDL dans l'interface et non dans la classe d'implémentation.

Cette annotation peut être utilisée sur une classe (comme dans l'exemple précédent) ou sur l'interface métier dite « endpoint ». Dans le deuxième cas, il faut impérativement indiquer l'interface utilisée *via* l'attribut `endpointInterface` dans la classe d'implémentation.

La seconde annotation, `@WebMethod`, est utilisée pour déclarer les méthodes accessibles par le service web (on parle d'opération dans le protocole SOAP). Voici les détails des attributs qu'elle contient :

- `operationName` : définit la valeur de l'attribut `name` de la balise `<operation>` dans le fichier WSDL. Celui-ci représente le nom de l'opération.
- `exclude` : marque une méthode comme étant non disponible par le service web. Ce paramètre est utilisé lorsque l'on souhaite masquer une méthode héritée, par exemple.

L'annotation `@WebParam`, quant à elle, permet de préciser des informations concernant les arguments des opérations définies dans le fichier WSDL. Voici le détail des attributs disponibles :

- `name` : définit le nom du paramètre qui sera utilisé dans le fichier WSDL. Par défaut, le nom de l'argument (du code Java) est utilisé.
- `header` : indique si la définition du paramètre se trouve dans l'entête (*header*) plutôt que dans le corps (*body*). La valeur par défaut étant `false`, la définition se trouve donc dans le corps.
- `mode` : indique si le paramètre est utilisé en entrée, en sortie ou les deux. Pour spécifier ce type, il faut utiliser l'énumération `WebParam.Mode` (valeurs : `IN`, `OUT`, `BOTH`).
- `targetNamespace` : définit le *namespace* à utiliser. Par défaut, on utilise un *namespace* vide. Cet attribut est à utiliser si le paramètre est *mappé* dans l'entête.

Toutes ces annotations font référence au fichier WSDL. Voici le fichier associé qui est auto-généré par le serveur d'applications :

```
<definitions name='WebServiceAccessBeanService' targetNamespace='http://
bean.session.stockmanager.labosun.com/jaws' xmlns='http://schemas.xmlsoap.org/
wsdl/' xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/' xmlns:tns='http://
bean.session.stockmanager.labosun.com/jaws' xmlns:xsd='http://www.w3.org/2001/
XMLSchema'>
 <types>
  <schema elementFormDefault='qualified' targetNamespace='http://
bean.session.stockmanager.labosun.com/jaws' xmlns='http://www.w3.org/2001/
XMLSchema' xmlns:soap11-enc='http://schemas.xmlsoap.org/soap/encoding/'
xmlns:tns='http://bean.session.stockmanager.labosun.com/jaws' xmlns:xsi='http://
www.w3.org/2001/XMLSchema-instance'>
   <complexType name='loginUser'>
    <sequence>
     <element name='userName' nillable='true' type='string'/>
```

```
      <element name='password' nillable='true' type='string'/>
     </sequence>
    </complexType>

    <complexType name='loginUserResponse'>
     <sequence>
      <element name='result' nillable='true' type='string'/>
     </sequence>
    </complexType>
    <element name='loginUser' type='tns:loginUser'/>
    <element name='loginUserResponse' type='tns:loginUserResponse'/>
   </schema>
  </types>

  <message name='WebServiceAccessBean_loginUserResponse'>
   <part element='tns:loginUserResponse' name='result'/>
  </message>
  <message name='WebServiceAccessBean_loginUser'>
   <part element='tns:loginUser' name='parameters'/>
  </message>
  <portType name='WebServiceAccessBean'>
   <operation name='loginUser'>
    <input message='tns:WebServiceAccessBean_loginUser'/>

    <output message='tns:WebServiceAccessBean_loginUserResponse'/>
   </operation>
  </portType>
  <binding name='WebServiceAccessBeanBinding' type='tns:WebServiceAccessBean'>
   <soap:binding style='document' transport='http://schemas.xmlsoap.org/soap/
http'/>
   <operation name='loginUser'>
    <soap:operation soapAction=''/>
    <input>
     <soap:body use='literal'/>

    </input>
    <output>
     <soap:body use='literal'/>
    </output>
   </operation>
  </binding>
  <service name='WebServiceAccessBeanService'>
   <port binding='tns:WebServiceAccessBeanBinding'
name='WebServiceAccessBeanPort'>
    <soap:address location='http://localhost:8080/StockManagerEJB/
WebServiceAccessBean'/>

   </port>
  </service>
</definitions>
```

Ce fichier représente, de façon standardisée, l'interface du service web déployé. Nous ne rentrons pas dans les détails des différentes balises de fichier. La plus importante est sans doute `<operation>` qui déclare une méthode proposée par le service.

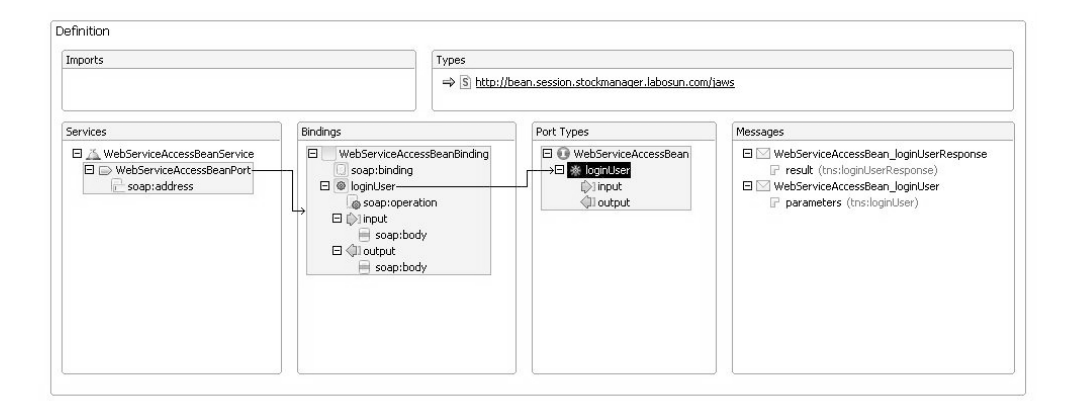


Figure 3.9 — Visualisation graphique du fichier WSDL

Dans l'exemple précédent, l'opération « loginUser » est liée la méthode `login()` implémentée dans la classe `WebServiceAccessBean`, comme on peut le voir dans le récapitulatif graphique du WSDL (fig. 3.9).

3.5 PACKAGING ET DÉPLOIEMENT

Une fois les différents EJB session conçus et développés, il reste des étapes supplémentaires avant de passer à leur utilisation : packaging et déploiement.

3.5.1 Packaging

Comme vu précédemment, il est possible d'intégrer des informations de structure des EJB au sein même du code source au moyen des annotations. Ces informations étaient renseignées dans le but d'informer le conteneur d'EJB, et plus généralement l'application, des composants mis à sa disposition et de leurs paramétrages. Il est intéressant de noter que ces annotations ne sont pas obligatoires. Tout comme dans les précédentes versions des spécifications EJB, il est possible d'intégrer ces informations structurelles au sein d'un fichier de configuration : le descripteur de déploiement.

Il convient donc de placer ceux-ci dans une archive « ejb-jar ». C'est cette archive qui sera utilisable par une application Java EE et consistera en une « brique » de celle-ci.

Un fichier de type « ejb-jar » doit contenir les fichiers suivants :

- Toute classe d'EJB (Session Bean, Entity Bean, Message-driven Bean)
- Toute interface d'EJB et interface de service web
- Toute classe d'intercepteur

- Toute classe correspondant à la clé primaire d'un EJB entité
- Le descripteur de déploiement des EJB dans le fichier « META-INF/ejb-jar.xml ». Ce fichier XML contient la configuration des différents EJB et permettra leur exploitation au sein de l'application. Il est toutefois non obligatoire.

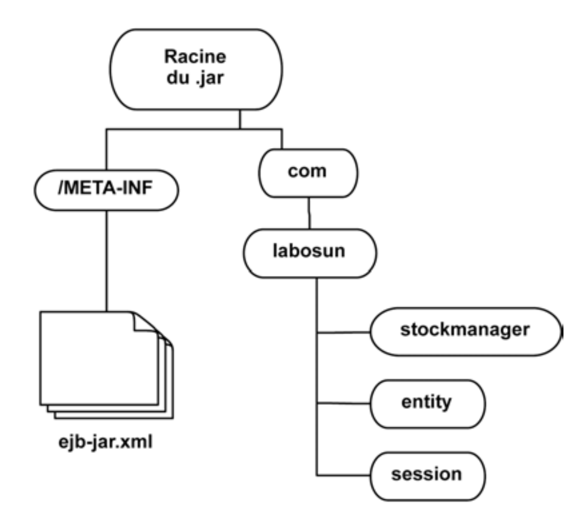


Figure 3.10 — Arborescence d'un EJB-JAR

L'ensemble des classes utilisées par les composants doivent être placées dans un fichier jar ayant l'extension « **.jar** ».

Du point de vue du développeur des EJB session, le descripteur de déploiement n'a pas à contenir d'informations relatives à leur définition dans le cas où les annotations seraient utilisées. Il permet cependant la surdéfinition de paramètres (variable d'environnement, références...).

3.5.2 Déploiement

La phase de déploiement pour le développeur est généralement très simple. Il suffit de placer l'archive packagée dans un répertoire spécial du serveur d'applications ou d'utiliser l'administration de celui-ci.

- JBoss requiert simplement que l'archive soit mise dans le dossier « deploy » de la configuration courante (`${jboss.home}/server/{configuration}/deploy`
- Glassfish permet le déploiement *via* l'interface d'administration web ou la ligne de commande d'administration ou encore *via* un dossier « autodeploy ».

La tâche est toutefois plus ardue pour le serveur d'applications. Celui-ci doit lire le contenu de l'archive, scanner les classes (afin de détecter les composants anno-

tés), mapper les Entity Beans à la base de données, définir la politique de sécurité (si elle est déclarée), générer les classes (proxy…) suivant les spécificités du fournisseur.

En résumé

L'évolution des Session Beans est plus qu'importante. Des procédures longues et laborieuses d'EJB 2, les nouvelles spécifications ont fait d'EJB 3 un système simple et intuitif. Tout est désormais fait grâce aux annotations.

Plus besoin de créer toutes les interfaces chères à EJB 2 ou d'écrire le descripteur de déploiement ; quelques annotations dans la classe du Bean s'en chargeront.

De même, la gestion du cycle de vie des Session Beans en devient plus que triviale et personnalisable, grâce aux méthodes *callbacks* et aux *interceptors*.

4
Les Entity Beans

Objectif

Au sein d'une architecture Java EE, les EJB sont utilisés pour créer les services. Cette technologie ne s'arrête cependant pas à cette couche, mais permet aussi de créer l'abstraction de l'accès aux données. Ce sont les EJB Entity qui rempliront cette fonction.

Tout comme les Session Beans, entre le client et la logique métier, les Entity Beans forment la passerelle entre la logique applicative et les sources de données. Ils permettent une abstraction quasi complète du stockage des données, permettant à l'application de rendre persistantes ou de charger des données de manière totalement transparente.

Après une présentation générale du concept des Entity Beans, nous exposerons, de manière pratique, la création de ceux-ci avec EJB 2, puis avec EJB 3.

4.1 RÔLE DES ENTITY BEANS

4.1.1 Qu'est-ce qu'un Entity Bean ?

Les Entity Beans ont été créés pour simplifier la gestion des données au niveau d'une application, mais aussi pour faciliter la sauvegarde en base de données. Plus concrètement, ces Entity Beans vous permettent de prendre en charge la persistance des données de votre application dans une ou plusieurs sources de données, tout en gardant les relations entre celles-ci. Ces composants établissent donc la relation entre votre application et vos bases de données. Contrairement aux Session Beans, les données d'un Entity Bean sont conservées, même après l'arrêt de l'application.

Les données de l'application sont typiquement : des utilisateurs, des factures, des produits, des adresses… Dans le monde Java, et plus généralement dans le monde objet, il est commun d'utiliser des classes pour chacun des types d'objets utilisés. On parle souvent d'objet métier ou entité (*Entity*) pour représenter les caractéristiques de ces objets.

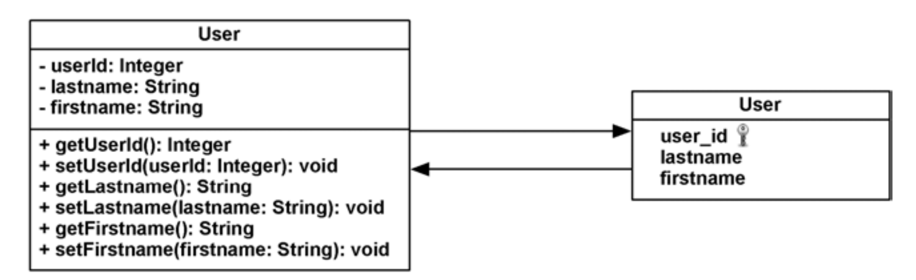


Figure 4.1 — Mécanisme de persistance objet/relationnel

La liaison entre les données et l'application par un objet s'appelle le *mapping* (relier). La figure 4.1 présente le *mapping* entre la table User et la classe `User`. On parle de mapping objet/relationnel[1] lorsque l'on connecte, de cette manière, une base de données relationnelle avec une application objet.

> **Attention :** l'utilisation des Entity Beans permet de représenter une entité de l'application et non une fonctionnalité. Par exemple, `User` serait un Entity Bean, mais `UserSubscription` serait un Session Bean ; un utilisateur étant voué à rester persistant, alors que l'inscription de l'utilisateur est une opération.

Contrairement aux Session Beans, les données des Entity Beans ont généralement une durée de vie longue ; elles sont enregistrées et stockées dans des systèmes de persistance (base de données).

4.1.2 Propriétés d'un Entity Bean

En abrégeant, il est possible de dire que l'objet à rendre persistant, *via* un mapping objet/relationnel, correspond à une table. Chaque propriété de cet objet est liée à un champ de la table. Chaque instance de cet objet représente généralement un enregistrement de la table. Toutefois, il est possible qu'un Entity Bean soit réparti sur plusieurs tables.

À l'inverse, une table dans une base de données regroupe un ensemble de champs. Ces champs représentent soit des informations directement liées à la table,

1. Mapping objet/relationnel : liaison automatique des propriétés d'un objet et d'une table de données permettant de sauvegarder/recharger facilement les informations d'une instance de l'objet.

soit des liens vers d'autres tables. Toutes ces informations ne sont pas pour autant des propriétés directes de l'objet en question.

Les Entity Beans suivent le même principe et doivent tous posséder un identifiant unique (clé primaire). Un Entity Bean `AccountInfo`, par exemple, peut être identifié à partir de son numéro de compte `accountId`. Cet identifiant unique permet à l'application de retrouver les données de l'Entity Bean associé.

Un Entity Bean `User` est, par exemple, caractérisé par des propriétés telles que : « nom », « prénom », « email », « téléphone »... On appelle ces propriétés : champs persistants. Ces propriétés sont *mappées* (liées) aux champs de la table associée. Durant l'exécution du programme, le conteneur EJB synchronise automatiquement l'état de ces propriétés avec la base de données.

Comme une table, un Entity Bean possède des champs relationnels. Toutefois, une grande différence existe. Un champ relationnel est représenté, dans un Entity Bean, par une propriété dont le type est un autre Entity Bean (fig. 4.2). On parle d'agrégation, en programmation objet. À l'opposé, une table est liée à une autre table par une clé étrangère (fig. 4.3).



Figure 4.2 — Liaison entre les classes User et AccountInfo



Figure 4.3 — Liaison entre les tables User et AccountInfo

Il existe quatre relations possibles entre les Entity Beans :

- **One To One** (un à un) : si un utilisateur ne peut avoir qu'un seul et unique compte alors la relation entre l'utilisateur et son compte est de type « One To One ».
- **One To Many** (un à plusieurs) et **Many To One** (plusieurs à un) : un utilisateur peut avoir plusieurs portefeuilles alors qu'un portefeuille est détenu par un seul utilisateur. La relation entre Portefeuille et Utilisateur est de type « Many To One » et la relation entre Utilisateur et Portefeuille est de type « One To Many ».

- **Many To Many** (plusieurs à plusieurs) : un utilisateur a plusieurs loisirs (*hobby*) et un loisir peut être partagé avec plusieurs utilisateurs. Dans ce cas, la relation est de type « Many To Many » entre utilisateur et utilisateur.

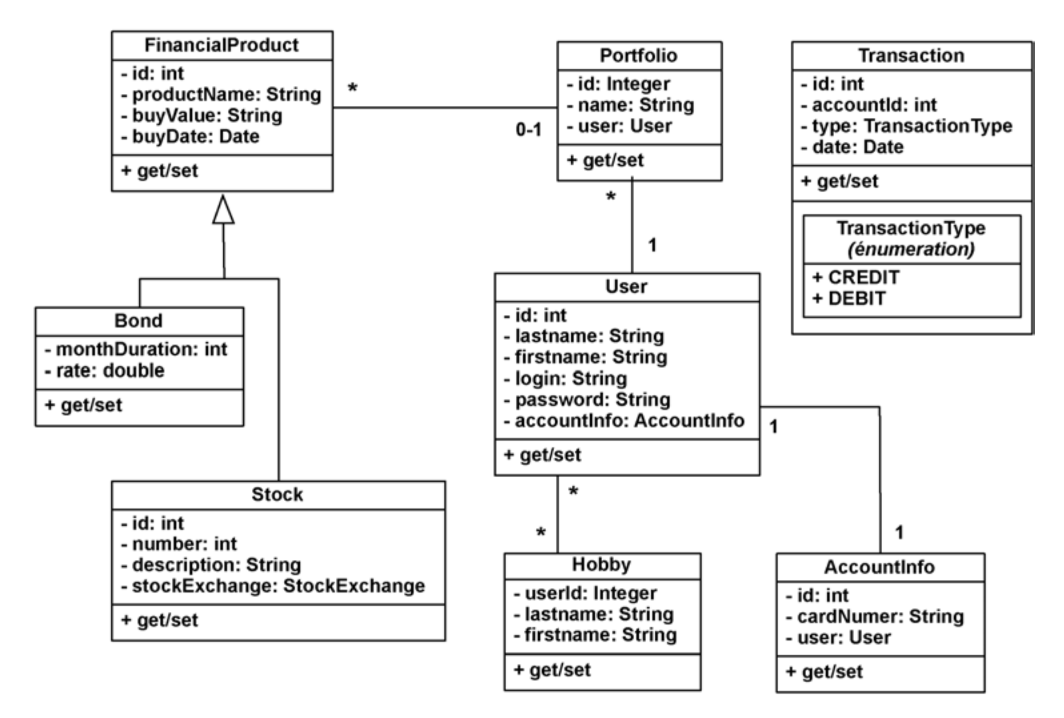


Figure 4.4 — Diagramme des entités de l'application exemple

Dans l'application d'exemple de la figure 4.4, un utilisateur possède plusieurs portefeuilles d'actions. Cette liaison est représentée par une relation « One To Many ».

4.1.3 Avantages des Entity Beans

Les Entity Beans sont exécutés dans le conteneur EJB, qui apporte aux développeurs de nombreux services facilitant le travail du développeur.

Le principal service est, bien entendu, la gestion de la persistance qui gère l'ensemble des accès aux données dans la mémoire ou dans les sources de données. L'utilisation de ce service, et donc des Entity Beans, procure de multiples avantages comparés à l'accès direct à la base de données. Cette solution apporte un mécanisme simple pour l'accès et la modification des données. En effet, il est plus facile, pour modifier le prénom d'un utilisateur au sein de votre application, d'appeler la méthode `User.setFirstName()` que d'exécuter une requête SQL brute. De plus, les Entity Beans étant standardisés, votre code est clair et plus facilement réutilisable (la définition d'un utilisateur se retrouve, par exemple, dans la majorité des applications).

Un Entity Bean étant un EJB, il hérite des services tels que la gestion des transactions (voir chapitre 9), de la sécurité… ainsi que de nombreux outils de développement optimisés pour leur création.

4.2 EJB 2 : ÉCRITURE D'UN ENTITY BEAN CMP

Ce chapitre détaille le processus de développement des Entity Beans dans la version 2 des EJB. Nous ne traitons cependant que le cas des CMP (*Contained Managed Persistence*) où la persistance est gérée automatiquement par le conteneur. Il existe, en effet, un mode appelé BMP (*Bean Managed Persistence*) qui permet d'écrire « manuellement » toute cette gestion.

Un Entity Bean CMP n'est pas une simple classe Java et impose la création d'un ensemble d'éléments : une classe, les interfaces fabrique (*home*) et métier (*business*), et une déclaration dans le descripteur de déploiement « ejb-jar.xml ».

> **Remarque :** nous nous baserons principalement sur deux entités de notre application d'exemple, dans les parties suivantes : `User` et `Portfolio` (respectivement : utilisateur et portefeuille).

4.2.1 La classe du Bean

Dans le modèle CMP 2.x, l'état des Entity Beans est automatiquement géré par le conteneur. Celui-ci a, de ce fait, la responsabilité de générer le code lié à l'Entity Bean, lors du déploiement. Le développeur, quant à lui, doit décrire les attributs et les relations par des champs persistants et relationnels **virtuels**. Ils sont qualifiés de virtuels car le développeur ne déclare pas ces champs directement mais utilise des accesseurs (*getters and setters*) **abstraits** qui doivent être déclarés dans la classe du Bean. C'est à partir de cette classe que le conteneur génère la classe d'implémentation de celui-ci. Cette phase varie en fonction des techniques employées par le conteneur.

```
public abstract class UserBean implements javax.ejb.EntityBean {
  public void setEntityContext(EntityContext ctx) throws EJBException,
    RemoteException {
  }
  public void unsetEntityContext() throws EJBException, RemoteException {
  }
  public void ejbRemove() throws RemoveException, EJBException,
    RemoteException {
  }
  public void ejbActivate() throws EJBException, RemoteException {
  }
  public void ejbPassivate() throws EJBException, RemoteException {
  }
  public void ejbLoad() throws EJBException, RemoteException {
  }
  public void ejbStore() throws EJBException, RemoteException {
  }
```

```
    public Integer ejbCreate(User user, AccountInfoLocal accountInfo,
      AddressLocal address) throws CreateException {
      setFirstName(user.getFirstName());
      setLastName(user.getLastName());
      setLogin(user.getLogin());
      setPassword(user.getPassword());
      return null;
    }
    public void ejbPostCreate(User user, AccountInfoLocal accountInfo,
      AddressLocal address){
      setAccountInfo(accountInfo);
      setAddress(address);
    }
    public abstract void setUserId(Integer id);
    public abstract Integer getUserId();
    public abstract void setFirstName(String firstName);
    public abstract String getFirstName();
    public abstract void setLastName(String lastName);
    public abstract String getLastName();
    public abstract void setLogin(String login);
    public abstract String getLogin();
    public abstract void setPassword(String password);
    public abstract String getPassword();
    public abstract java.util.Collection getPortfolios();
    public abstract void setPortfolios(java.util.Collection portfolios);
    //...
  }
```

L'Entity Bean `User` définit les champs persistants suivants : `userId`, `firstName`, `lastName`, `login`, `password` et le champ relationnel `portfolios`. Il est ici impératif d'utiliser des accesseurs virtuels. Si vous déclarez des variables d'instances (propriétés), elles ne seront pas considérées comme des champs persistants.

La classe abstraite `UserBean` implémente l'interface `javax.ejb.EntityBean` qui indique au conteneur que la classe représente un Entity Bean. Elle définit également des méthodes de gestion du cycle de vie de l'Entity Bean (fig. 4.5).

Les méthodes `ejbLoad()`, `ejbStore()` et `ejbRemove()` ont pour vocation d'interagir avec la base de données. Elles permettent respectivement de charger, d'enregistrer les données, et de supprimer l'enregistrement. Ces méthodes sont peu utilisées en général car le conteneur s'occupe de gérer la synchronisation avec la base de données. Elles s'avèrent, cependant, intéressantes lorsque vous souhaitez effectuer des opérations avant la mise à jour de la base de données (`ejbStore()`) ou après le chargement des données (`ejbLoad()`). Imaginons que votre Entity Bean utilise des champs binaires (exemple : une image) ou de texte à capacité importante (exemple : un article) ; il sera peut-être intéressant de (dé)compresser ces données. Vous pouvez alors utiliser ces méthodes pour effectuer vos opérations.

Les méthodes `ejbPassivate()` et `ejbActivate()` servent à écrire sur disque l'état de l'entité persistante lorsqu'elle n'est plus utilisée depuis un certain temps. Ces méthodes sont peu utilisées dans la pratique car le conteneur gère très bien cela, dans la plupart des cas.

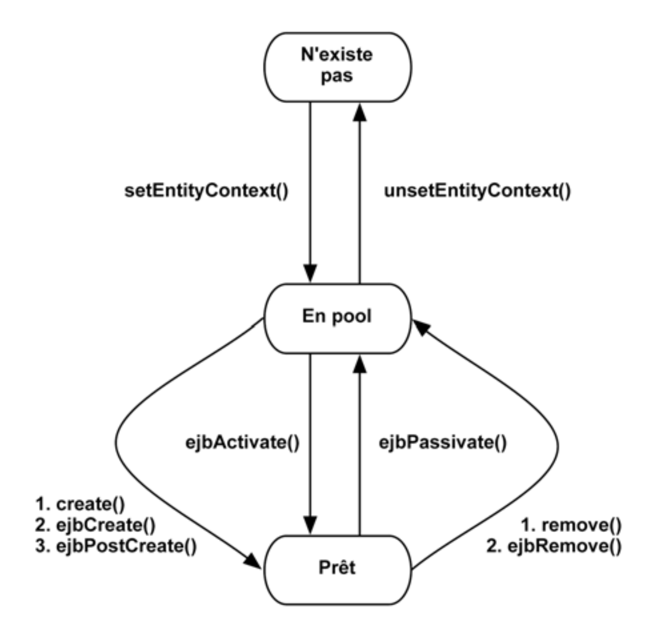


Figure 4.5 — Cycle de vie d'un Entity Bean

Les méthodes `setEntityContext()` et `unsetEntityContext()` permettent d'informer le Bean lors de sa création et de sa destruction. Lors de l'appel de `setEntityContext()`, le développeur doit se charger de récupérer les ressources qui ne sont pas spécifiques à l'Entity Bean (fabrique de connexion JMS, datasource…). L'appel de `unsetEntityContext()` indique la fin de vie du Bean ; c'est le bon moment pour libérer les ressources chargées lors de l'appel de `setEntityContext()`.

Attention : ces méthodes doivent obligatoirement exister. Cependant, l'appel de celles-ci est uniquement à la charge du conteneur.

Les méthodes `ejbCreate()` et `ejbPostCreate()` sont obligatoires, sans pour autant être définies dans l'interface `EntityBean`. Elles servent à définir la façon de créer une nouvelle instance de l'Entity Bean.

Il est possible de surcharger la méthode `ejbCreate()` si l'on souhaite avoir différentes façons de créer une entité. Le type de retour de la méthode `ejbCreate()` doit être le type de la clé primaire (dans l'exemple, la clé primaire est le champ persistant `userId` de type `Integer`). Néanmoins, pour chaque méthode `ejbCreate()` présente, une méthode `ejbPostCreate()` ayant les mêmes arguments doit exister. La méthode `ejbCreate()` est appelée par le conteneur lorsqu'une application cliente souhaite créer une nouvelle entité persistante. Cette méthode doit se charger d'affecter (*via* les *setters*) les valeurs pour chaque champ persistant. Une fois cette méthode exécutée, le conteneur appelle la méthode `ejbPostCreate()` correspondante qui s'occupe, quant à elle, d'affecter les champs relationnels.

> **Remarque :** la méthode `ejbPostCreate` doit retourner « null », d'après la spécification 2.0, afin de faciliter l'évolution d'un EJB BMP vers un EJB CMP.

Le tableau 4.1 récapitule les différents accès en base correspondant aux méthodes du cycle de vie d'un Entity Bean.

Tableau 4.1 — Récapitulatif des accès en base de données selon le cycle de vie d'un Entity Bean

Méthode	Requête SQL
`ejbCreate()`	INSERT
`ejbLoad()`	SELECT
`ejbRemove()`	DELETE
`ejbStore()`	UPDATE

Comme vous pouvez le constater, la création de la classe d'un Entity Bean CMP n'est pas intuitive. Il est préférable de bien connaître le fonctionnement du conteneur pour en comprendre les différents aspects.

4.2.2 Les interfaces

Le principe et la mise en place des interfaces sont les mêmes que ceux des Session Beans. Un Entity Bean peut être accessible en local et à distance suivant les interfaces qui lui ont été assignées. L'utilisation et le fonctionnement de celles-ci diffèrent légèrement selon le cas.

> **Remarque :** l'utilisation d'un Entity Bean à distance n'est pas une bonne pratique J2EE. En effet, les performances sont réduites car chaque appel d'un accesseur doit faire l'objet d'une communication réseau. De plus, vous ne pouvez utiliser facilement les relations entre Entity Beans lorsqu'ils sont utilisés à distance.

Interface « home »

L'interface de fabrication des EJB, plus communément appelée l'interface *home* (ou *local home* pour un accès local) n'est pas utilisée uniquement pour la création de l'Entity Bean en base de données. Elle sert également à récupérer une ou plusieurs instances de l'Entity Bean *via* des méthodes de recherches : les « *finders* ». Tout Entity Bean CMP possède au moins la méthode `findByPrimaryKey()` qui permet de récupérer une instance de l'entité à partir de sa clé primaire.

Dans le cas d'un Entity Bean CMP, chaque *finder* correspond à une méthode dans l'interface home. Cette méthode n'a pas de correspondance directe avec la classe du

Bean, mais est liée à une requête EJB-QL[1]. Le code de la méthode de recherche est à la charge du conteneur qui s'en occupe lors de la phase du déploiement.

```
public interface UserLocalHome extends javax.ejb.EJBLocalHome {

   public UserLocal create(User user , AccountInfoLocal accountInfo ,
     AddressLocal address) throws javax.ejb.CreateException;

   public UserLocal findByPrimaryKey(Integer pk) throws
     javax.ejb.FinderException;

   public Collection findAll() throws javax.ejb.FinderException;

}
```

Cette interface hérite de `EJBLocalHome` car elle définit la fabrique locale de l'entité. L'interface déclare la méthode `create()` suivant les arguments de `ejbCreate()` de la classe `UserBean`. Cependant, à la différence de celle-ci qui retourne la clé primaire (de type `Integer`), la méthode `create()` retourne un objet de type `UserLocal`. C'est l'interface métier de l'entité `User`. La correspondance entre l'interface et la classe du Bean est automatiquement faite par le conteneur.

Interface « business »

Contrairement à l'interface *home* qui travaille sur l'ensemble des instances d'un Entity Bean, l'interface métier (ou *business*) travaille sur une instance précise.

Les méthodes métiers, définies dans cette interface, sont généralement les méthodes d'accès aux données de l'Entity Bean (accesseurs). Le développeur a la charge de définir les méthodes qui sont accessibles par l'application cliente.

```
public interface UserLocal extends javax.ejb.EJBLocalObject {
   public void setUserId( java.lang.Integer id ) ;
   public java.lang.Integer getUserId(  ) ;
   public void setFirstName( java.lang.String firstName ) ;
   public java.lang.String getFirstName(  ) ;
   public void setLastName( java.lang.String lastName ) ;
   public java.lang.String getLastName(  ) ;
   public void setLogin( java.lang.String login ) ;
   public java.lang.String getLogin(  ) ;
   public void setPassword( java.lang.String password ) ;
   public java.lang.String getPassword(  ) ;
   public AddressLocal getAddress(  ) ;
   public void setAddress( AddressLocal address ) ;
   public AccountInfoLocal getAccountInfo(  ) ;
   public void setAccountInfo( AccountInfoLocal accountInfo ) ;
   public java.util.Collection getPortfolios(  ) ;
   public void setPortfolios( java.util.Collection portfolios ) ;
   public com.labosun.stockmanager.entity.User getData(  ) ;
}
```

1. EJB-QL (*Entreprise Java Bean Query Language*) : langage de requête utilisé dans les Entity Beans CMP. Une explication plus détaillée est présentée au chapitre 8.

Cette interface ressemble de très près à la définition d'une classe `User`. C'est effectivement le cas. Un problème récurrent en EJB 2 est l'utilisation d'un Entity Bean par une application cliente. En effet, les appels de méthodes métiers sont coûteux en temps et en performance. Pour pallier à cela, les développeurs utilisent le *design pattern* : « Value Object » (objet de valeur). Ce *design pattern* consiste en l'élaboration d'une classe standard ayant les mêmes propriétés que l'Entity Bean tout en étant indépendant de la technologie EJB. Voici un exemple de « Value Object » pour l'Entity Bean `User` :

```
public class User implements Serializable {
  private int id;
  private String lastName;
  private String firstName;
  private String login;
  private String password;
  private Collection portfolios;

  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getLastName() {
    return lastName;
  }
  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
  public String getFirstName() {
    return firstName;
  }
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }
  public String getLogin() {
    return login;
  }
  public void setLogin(String login) {
    this.login = login;
  }
  public String getPassword() {
    return password;
  }
  public void setPassword(String password) {
    this.password = password;
  }
  public Collection getPortfolios() {
    return portfolios;
  }
  public void setPortfolios(Collection portfolios) {
    this.portfolios = portfolios;
  }
}
```

Cette classe ressemble étrangement à l'implémentation de l'interface métier. Cependant aucune dépendance vers cette interface ou vers une autre technologie n'est utilisée. Cette classe permet au développeur de « détacher » les valeurs d'une instance de l'Entity Bean User. L'application cliente peut alors travailler avec cette classe sans pour autant être connectée à l'EJB.

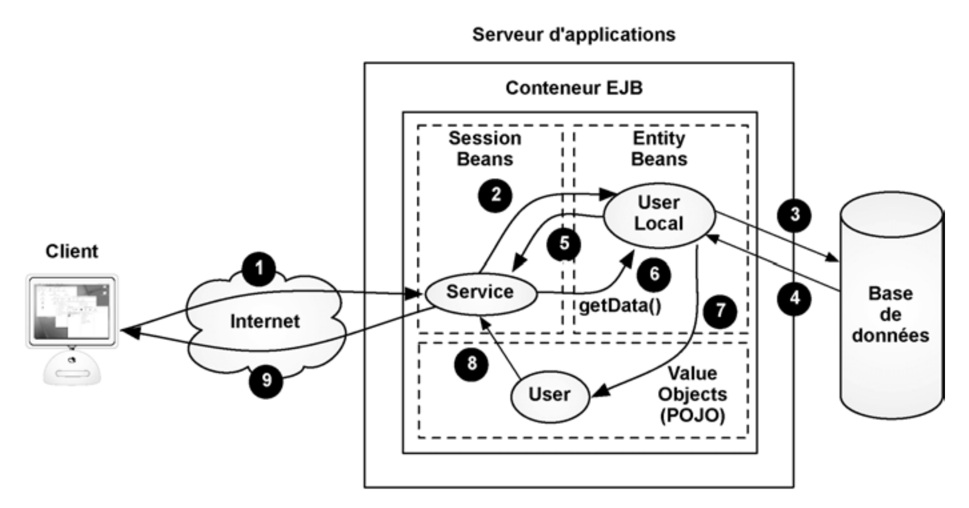


Figure 4.6 — Principe du « Value Object »

Vous pouvez remarquer (fig. 4.6) que le client doit toujours travailler avec l'interface *home* pour récupérer (numéro 2) les instances de UserLocal (numéro 6 : interface métier). Cette interface métier définit la méthode getData() qui permet de faire la conversion entre UserLocal et la classe User (numéro 6). Le code de la méthode getData() peut ressembler à :

```
public User getData() {
  User user = new User();
  // données de l'utilisateur
  user.setFirstName(getFirstName());
  user.setLastName(getLastName());
  user.setId(getUserId());
  user.setLogin(getLogin());
  user.setPassword(getPassword());
  return user;
}
```

Ceci est un exemple simple de conversion. Cette méthode peut s'avérer plus complexe à mettre en place selon l'importance des relations entre les Entity Beans.

4.2.3 Le descripteur de déploiement

Le dernier élément à paramétrer est le descripteur de déploiement. Tout comme les Session Beans, les Entity Beans doivent être déclarés. Toutefois, ce type d'EJB possède beaucoup plus de paramètres. La grande différence réside dans le fait qu'il

faut définir toutes les propriétés du Bean qui doivent être persistantes ou relationnelles.

Déclaration de l'Entity Bean

La déclaration d'un Entity Bean se fait par la balise `<entity>`. Tout comme la balise `<session>`, elle doit être incluse dans le corps de la balise `<entreprise-beans>`. Voici la déclaration de l'entité `User` :

```
<entity>

  <!-- Mêmes balises que pour les Sessions Beans -->
  <description>
    <![CDATA[Déclaration de l'entité Utilisateur « User »]]>
  </description>
  <ejb-name>User</ejb-name>
  <local-home>
    com.labosun.stockmanager.ejb2.entity.interfaces.UserLocalHome
  </local-home>
  <local>
    com.labosun.stockmanager.ejb2.entity.interfaces.UserLocal
  </local>
  <ejb-class>com.labosun.stockmanager.ejb2.entity.ejb.UserBean</ejb-class>

  <!-- Balises spécifiques au Entity Bean -->
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>
  <reentrant>false</reentrant>
  <cmp-version>2.x</cmp-version>
  <abstract-schema-name>User</abstract-schema-name>
  <cmp-field>
    <description><![CDATA[Clé primaire de l'utilisateur]]></description>
    <field-name>userId</field-name>
  </cmp-field>
  <cmp-field>
    <description><![CDATA[Prénom de l'utilisateur]]></description>
    <field-name>firstName</field-name>
  </cmp-field>
  <cmp-field>
    <description><![CDATA[Nom de l'utilisateur]]></description>
    <field-name>lastName</field-name>
  </cmp-field>
  <cmp-field>
    <description><![CDATA[Login de l'utilisateur]]></description>
    <field-name>login</field-name>
  </cmp-field>
  <cmp-field>
    <description><![CDATA[Mot de passe de l'utilisateur]]></description>
    <field-name>password</field-name>
  </cmp-field>
  <primkey-field>userId</primkey-field>

  <query>
    <query-method>
      <method-name>findAll</method-name>
      <method-params>
```

```
      </method-params>
    </query-method>
    <result-type-mapping>Local</result-type-mapping>
    <ejb-ql><![CDATA[SELECT Object(u) FROM User u]]></ejb-ql>
  </query>
</entity>
...
```

Les balises précisant la classe, les interfaces, le nom et la description d'un Entity Bean sont les mêmes que celles utilisées avec les Session Beans.

Les balises supplémentaires sont liées à la persistance. Dans la définition de l'entité, nous retrouvons tous les champs persistants, déclarés par les accesseurs dans `UserBean` (userId, firstname, lastname, login, password). L'élément `<persistence-type>` indique si la persistance est prise en charge par le conteneur (`Container`) ou par le composant (`Bean`).

La balise `<prim-key-class>` indique la classe de la clé primaire (ici `Integer`). Cette balise va de pair avec `<primkey-field>` qui désigne le champ persistant à utiliser comme clé primaire (ici `userId`).

La balise `<reentrant>`, quant à elle, indique si l'Entity Bean accepte la « réentrance » ou non (valeur par défaut). La réentrance se produit lorsqu'un Bean A accède à un Bean B et que ce Bean B accède au Bean A (fig. 4.7). La spécification EJB déconseille l'activation de la réentrance pour des raisons de sécurité de code et de corruptions de données.

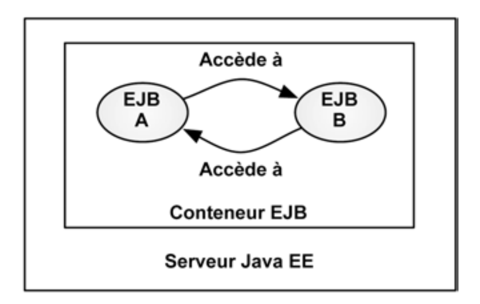


Figure 4.7 — Concept de réentrance

La balise `<abstract-schema>` définit le nom abstrait de l'entité. Ce nom est utilisé dans les requêtes EJB-QL pour référencer l'entité. Il est utilisé dans la balise `<query>` de l'exemple, qui permet de déclarer une requête EJB-QL associée à une méthode de l'interface *home*. Ici, la méthode `findAll()` retourne l'ensemble des utilisateurs (voir chapitre 7).

Relations entre entités

La définition d'une relation entre deux entités est séparée de la définition des Entity Beans. Le fichier « ejb-jar.xml » contient une section spécifique, nommée

<relationship>, pour y définir ces relations, *via* la balise <ejb-relation>. Voici la description de la relation entre User et Portfolio :

```
...
<relationships>
...
<ejb-relation>
   <ejb-relation-name>portfolio-user</ejb-relation-name>

   <ejb-relationship-role>
      <ejb-relationship-role-name>portfolio-user</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <cascade-delete/>
      <relationship-role-source >
         <ejb-name>Portfolio</ejb-name>
      </relationship-role-source>
      <cmr-field >
         <cmr-field-name>user</cmr-field-name>
      </cmr-field>
   </ejb-relationship-role>

   <ejb-relationship-role>
      <ejb-relationship-role-name>user-portfolio</ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
         <ejb-name>User</ejb-name>
      </relationship-role-source>
      <cmr-field>
         <cmr-field-name>portfolios</cmr-field-name>
         <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
   </ejb-relationship-role>

</ejb-relation>

</relationships>
...
```

Voici la correspondance dans la classe UserBean :

```
//...
   public AddressLocal getAddress(  ) ;
   public void setAddress( AddressLocal address ) ;
   public AccountInfoLocal getAccountInfo(  ) ;
   public void setAccountInfo( AccountInfoLocal accountInfo ) ;
   public java.util.Collection getPortfolios(  ) ;
   public void setPortfolios( java.util.Collection portfolios ) ;
//...
```

Toute relation a un nom qui est défini par la balise <ejb-relation-name>.

La déclaration des liaisons entre les deux entités est réalisée par la balise <ejb-relationship-role>. Elle définit le nom, la cardinalité, l'entité source et le champ relationnel utilisé. Ici, la relation entre User et Portfolio est définie par le fait qu'un utilisateur peut avoir plusieurs portefeuilles et qu'un portefeuille n'est lié qu'à un utilisateur. Il existe donc deux côtés de la relation « Portfolio-User » et « User-

Portfolio ». La classe `User` contient une propriété `portfolios` (une Collection de `PortfolioLocal`) alors que la classe `Portfolio` contient une propriété `user` (de type `UserLocal`).

Le développeur doit définir les deux côtés de la relation pour que le conteneur puisse déterminer la cardinalité de la relation entre les deux entités (*One To One*, *One To Many*…).

4.2.4 Conclusion

Nous avons vu, dans cette partie, la laborieuse mise en place d'un Entity Bean CMP avec les spécifications EJB 2. Bien que rapide pour l'utilisation de quelques Entity Beans, elle devient de plus en plus difficile au fur et à mesure de l'ajout de relations avec d'autres Entity Beans. Il en est de même lors de la modification d'un ou plusieurs Entity Beans au sein d'une application existante.

4.3 EJB 3 : ÉCRITURE D'UN ENTITY BEAN

4.3.1 Cure de jeunesse

La spécification EJB 3 apporte de nombreux changements et simplifications pour le développement d'Entity Beans. Le changement principal est l'externalisation de la gestion de la persistance avec l'API Persistence 1.0 (JSR 220). Cette API est une réelle cure de jeunesse pour le développement d'Entity Beans. Elle fournit un modèle de persistance à base de POJO pour le mapping objet/relationnel.

La simplicité de développement (*Ease of development*) est en partie due à l'introduction des annotations et à la configuration par exception. En effet, nous allons voir que les spécifications définissent des comportements par défaut qui sont généralement corrects pour une majeure partie des applications.

L'autre grande nouveauté est la suppression des interfaces liées aux Entity Beans. Même si cela impose l'utilisation d'un Session Bean pour gérer l'accès aux données, les architectes s'en féliciteront ! En effet, il n'est pas possible d'accéder directement à l'Entity Bean depuis l'application cliente. Toutefois, les bonnes pratiques conseillaient d'éviter ces accès directs aux données.

Cette évolution est, avant tout, le fruit de la communauté. C'est par l'essor des *frameworks*, tels Hibernate, iBatis ou Toplink que le mapping objet/relationnel a pu se développer et finalement devenir un standard.

Comme nous l'avons vu précédemment avec les EJB 2, nous allons présenter et illustrer la création des éléments nécessaires à l'utilisation d'Entity Beans avec EJB 3. Nous détaillerons, cependant, beaucoup plus les différentes possibilités de chaque étape.

4.3.2 La classe de l'entité

La classe de l'Entity Bean est le premier élément à définir en EJB 3. Comme dit précédemment, les Entity Beans sont des POJO. De ce fait, la création d'un Entity Bean `User` se résume à la création d'une classe `User`, tel que nous le ferions dans une application Java SE. L'état persistant de l'entité est représenté par les variables d'instances de la classe (contrairement à l'abstraction utilisée en EJB 2) qui correspondent aux propriétés du POJO. Ces variables peuvent être privées (`private`), protégées (`protected`) ou non spécifiées. Les clients ne peuvent alors pas accéder directement aux variables et doivent utiliser les accesseurs (*getter and setter*) ou d'autres méthodes métiers de la classe.

Cette classe doit, cependant, respecter certaines règles :

- Cette classe peut être abstraite (`abstract`) ou concrète.
- La classe peut aussi bien hériter d'une classe entité que d'une classe non-entité, et inversement.
- Les méthodes, les propriétés de la classe et la classe elle-même ne doivent pas être finales.
- Si une instance de l'entité a la possibilité d'être envoyée à un client distant, alors la classe doit implémenter `java.io.Serializable`. En effet, RMI utilise la (dé)sérialisation pour passer les arguments entre l'application cliente et le serveur.
- La classe doit posséder un constructeur sans argument qui peut être public (`public`) ou protégé (`protected`). Néanmoins, la classe peut posséder des constructeurs surchargés si la condition précédente est respectée.

Remarque : le but du constructeur par défaut (sans argument) est de simplifier l'instanciation de la classe par le conteneur. La gestion dynamique de n'importe quel constructeur est une tâche lourde pour celui-ci sans pour autant être utile. La solution a donc été de définir un constructeur commun à tous les Entity Beans : le constructeur par défaut.

Les Entity Beans EJB 3 supportent l'héritage, les associations et les requêtes polymorphiques (voir chapitre 7). Ces différents concepts seront expliqués tout au long de ce chapitre.

```
@Entity
public class User {
//...
}
```

C'est grâce à l'annotation **`@Entity`** que le conteneur pourra savoir quelles classes il doit considérer en tant qu'Entity Bean. Cette annotation se situe au niveau de la classe et permet de définir, si besoin, le nom de l'entité *via* l'attribut `name`.

Le nom utilisé doit être unique dans une application. La valeur par défaut utilisée est le nom de la classe (« User » dans l'exemple précédent). Ce nom est utilisé pour

représenter l'entité dans les requêtes EJB-QL ; c'est le schéma abstrait de l'Entity Bean.

```
@Entity(name = "MyUser")
public class User {
//...
}
```

Un Entity Bean est lié à une table en base de données. Cette liaison s'appelle le *mapping*. Par défaut, un Entity Bean `User` est *mappé* sur la table « User ». Si pour certaines raisons, vous deviez le *mapper* sur une autre table, vous devrez alors utiliser l'annotation `@Table`. Cette annotation possède différents attributs :

- `name` (requis) : définit le nom de la table à utiliser pour le *mapping*.
- `catalog` (optionnel) : définit le catalogue utilisé.
- `schema` (optionnel) : définit le schéma utilisé.
- `uniqueConstraints` (optionnel) : définit les contraintes qui seront placées sur la table. Cet attribut est utilisé lorsque le conteneur génère les tables au déploiement et n'affecte en rien l'exécution même de l'entité.

```
@Entity
@Table(name = "User")
public class User {
//...
}
```

4.3.3 Les champs persistants

Un champ persistant représente une caractéristique d'un Entity Bean. Comme vu précédemment en EJB 2, un champ persistant est, pour une entité utilisateur par exemple : le nom, le prénom, l'adresse, la date de naissance, le sexe…

N'importe quel champ (variable d'instance) non « static » et non « transient », d'un Entity Bean, est automatiquement considéré comme persistant par le conteneur.

Un jeu d'annotations standard est défini dans la spécification EJB 3. On peut considérer deux types d'annotations liés au mapping objet/relationnel :

- les annotations liées aux propriétés,
- les annotations liées aux colonnes.

Ces deux types d'annotations, qui seront décrites de manière plus détaillée dans les parties suivantes, peuvent se placer de deux façons :

- directement sur les champs (type « FIELD »),
- sur les accesseurs (et plus précisément le *getter*, type « PROPERTY »).

```
@Entity
...
public class User implements Serializable {

  // Déclaration de l'énumération des sexes (Masculin / Féminin)
  public enum SexType { MALE, FEMALE };

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private int id;

  @Basic // optionnel
  private String lastName;

  private String firstName;

  @Column(unique=true)
  private String login;

  private String password;

  @Enumerated(value=EnumType.STRING)
  @Column(length=5)
  private SexType sex;

  @Temporal(TemporalType.DATE)
  private Date birthDate;
  //...
}
```

Les annotations étant ici précisées sur les variables d'instance (façon « FIELD »), le conteneur injecte les valeurs directement dans celles-ci.

Voici le même exemple, mais en utilisant la façon « PROPERTY » :

```
@Entity
...
public class User implements Serializable {
  // Déclaration de l'énumération des sexes (Masculin / Féminin)
  public enum SexType { MALE, FEMALE };

  private int id;
  private String lastName;
  private String firstName;
  private String login;
  private String password;
  private SexType sex;
  private Date birthDate;

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  public int getId() { return id; }

  public void setId(int id) { this.id = id; }

  @Basic // optionnel
  public String getLastName() { return lastName; }
```

```
  public void setLastName(String lastName) { this.lastName = lastName; }

  public String getFirstName() { return firstName; }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  @Column(unique=true)
  public String getLogin() { return login; }

  public void setLogin(String login) { this.login = login;}

  public String getPassword() { return password; }

  public void setPassword(String password) { this.password = password; }

  @Enumerated(value=EnumType.STRING)
  @Column(length=5)
  public SexType getSex() { return sex; }

  public void setSex(SexType sex) { this.sex = sex; }

  @Temporal(TemporalType.DATE)
  public Date getBirthDate() { return birthDate;}

  public void setBirthDate(Date birthDate) { this.birthDate = birthDate; }
  //...
}
```

En localisant les annotations sur les accesseurs, le conteneur injecte les valeurs *via* ceux-ci. Ce sont eux qui affectent ces valeurs aux variables d'instance.

Conseil : l'utilisation de l'une ou l'autre de ces méthodes dépend de ce que vous souhaitez faire avec votre Entity Bean. Toutefois, il est préférable d'utiliser les accesseurs afin d'avoir un meilleur contrôle des données injectées par le conteneur.

Nous utiliserons l'Entity Bean `User` comme exemple de référence afin d'expliquer les différentes annotations.

Annotations liées aux propriétés simples

Toutes les propriétés d'un Entity Bean étant vouées à être rendues persistantes peuvent être paramétrées, selon les besoins du développeur. Bien que le paramétrage par défaut soit souvent suffisant, il est parfois intéressant de préciser telle ou telle spécificité pour une meilleure optimisation.

Les annotations dédiées à ce paramétrage permettent de préciser au moteur de persistance les différentes informations relatives au type et aux contraintes des propriétés.

La politique de l'API Persistence est de considérer toute propriété comme un champ persistant. Cela signifie qu'il n'est pas nécessaire d'annoter les propriétés pour

les désigner persistantes. Le conteneur considère par défaut que la propriété est annotée avec `@Basic` avec les valeurs par défaut des attributs suivants :

- `fetch` (`FetchType.EAGER` par défaut) : définit si le contenu de la propriété doit être chargé à la demande[1] (`FetchType.LAZY`, « paresseusement », en anglais) ou au moment du chargement de l'entité (`FetchType.EAGER`, « désireux », en anglais).
- `optional` (`true` par défaut) : définit si la propriété accepte la valeur « null » ou non. Cet attribut ne fonctionne pas pour les types primitifs (qui ne peuvent être nuls).

Contrairement aux procédures EJB 2 où il faut préciser les champs à rendre persistants, le développeur doit, ici, inverser sa façon de penser et spécifier les champs non persistants au conteneur. Pour indiquer qu'une propriété ne doit pas être enregistrée, il faut annoter son *getter* avec `@Transient` (seulement si cette méthode est de la forme `getXxx()`).

Bien que certains champs soient d'un même type, il est souvent possible que ceux-ci n'aient pas les mêmes contraintes, en termes de taille, par exemple. Prenons le nom d'un utilisateur qui est une chaîne ne dépassant pas 40 caractères et le contenu d'un article qui est également une chaîne de caractères mais dont la longueur est variable et n'est pas représentée de la même façon en base de données. Le développeur utilise cependant, dans les deux cas, l'objet `String`. Pour différencier ces deux utilisations de `String`, le développeur peut utiliser l'annotation `@Lob` (*Large Binary Object*). Celle-ci s'avère également utile lorsque vous souhaitez stocker des tableaux de bytes (`byte[]` ou `Byte[]`) pour représenter le contenu d'un fichier par exemple.

> **Remarque :** l'annotation `@Lob` s'applique également sur des propriétés de type `java.sql.Clob` (*Character Large Object*) ou `java.sql.Blob` (*Binary Large Object*).

```
// utilisé pour les longs textes
@Lob
public String getArticleContent() {
  return articleContent;
}

// utilisé pour les fichiers
@Lob
@Basic(fetch=FetchType.LAZY)
public byte[] getUserPicture() {
  return userPicture;
}
```

Nous avons volontairement ajouté `@Basic(fetch=FetchType.LAZY)` dans le deuxième exemple. On utilise généralement le chargement à la demande (*lazy loading*, voir section 6.6.4) lorsque le contenu d'une propriété est de grande taille.

1. Chargement à la demande ou « Lazy Loading » : voir le paragraphe 7.6.4.

De la même manière, les types `java.util.Date` ou `java.util.Calendar` utilisés pour définir des propriétés dites « temporelles » peuvent être paramétrés pour spécifier le format le plus adéquat à sa mise en persistance. Ceci peut être précisé grâce à l'annotation `@Temporal` qui prend en paramètre un `TemporalType` (énumération) dont les valeurs sont les suivantes :

- `DATE` : utilisé pour la date (`java.sql.Date`),
- `TIME` : utilisé pour l'heure (`java.sql.Time`),
- `TIMESTAMP` : utilisé pour les temps précis (`java.sql.TimeStamp`).

```
@Temporal(TemporalType.DATE)
public Calendar getDateOfBirth() {
    return dateOfBirth ;
}
```

J2SE 5.0 apporte un nouveau type de données au langage Java : les énumérations. Ce type existe depuis plusieurs années au sein des bases de données. Il est enfin possible de les utiliser dans les Entity Beans. L'énumération permet de spécifier un ensemble de valeurs possibles pour une propriété. Par exemple, le sexe d'un `User` n'a que deux valeurs possibles : masculin (`MALE`) ou féminin (`FEMALE`). La solution idéale dans cette situation est bien entendu l'utilisation de l'énumération. La valeur d'une énumération peut être enregistrée soit *via* une chaîne de caractère soit *via* un entier. L'annotation `@Enumerated` prend en paramètre un objet `EnumType` qui définit la façon de stocker cette valeur. Les valeurs `EnumType.STRING` ou `EnumType.ORDINAL` sont utilisées respectivement pour l'enregistrement dans une chaîne de caractère ou dans un entier.

```
public enum SexType { MALE, FEMALE };
//...
@Enumerated(value=EnumType.STRING)
public SexType getSex() { return sex; }
```

Bien que les annotations de propriétés couvrent un ensemble important des types utilisés en général dans les applications, celles-ci sont destinées à être utilisées par le gestionnaire de persistance. Ces paramétrages n'ont donc pas forcément toujours l'impact voulu dans la base de données, lorsque celle-ci est générée automatiquement.

Annotations liées aux colonnes simples

D'autres annotations permettent de préciser le paramétrage des colonnes (dans la table relationnelle) liées aux propriétés persistantes. Grâce à celles-ci, il est alors possible de spécifier le type SQL, la longueur du champ, et de nombreuses autres propriétés à utiliser pour une propriété persistante.

C'est l'annotation `@Column` qu'il faudra utiliser pour préciser ces paramétrages SQL. Celle-ci « surdéfinit » les valeurs par défaut déclarées par la spécification EJB 3. Cette annotation peut s'utiliser conjointement avec les précédentes.

Voici une description des attributs, tous optionnels, de l'annotation `@Column` :

- `name` : précise le nom de la colonne liée. Le nom de la propriété est utilisé par défaut.
- `unique` : précise si la propriété est une clé unique ou non (la valeur est unique dans la table).
- `nullable` : précise si la colonne accepte des valeurs nulles ou non.
- `insertable` : précise si la valeur doit être incluse lors de l'exécution de la requête SQL INSERT. La valeur par défaut est `true`.
- `updatable` : précise si la valeur doit être mise à jour lors de l'exécution de la requête SQL UPDATE. La valeur par défaut est `true`.

> **Remarque :** les attributs `insertable` et `updatable` sont généralement utilisés lorsque la colonne est utilisée par plusieurs propriétés dans une même entité. Cela se produit, par exemple, lorsqu'une colonne est à la fois une clé primaire et étrangère.

- `columnDefinition` : précise le « morceau de code SQL » pour la définition de la colonne dans la base de données. C'est avec cet attribut que l'on peut préciser le type SQL de la colonne.
- `table` : précise la table utilisée pour contenir la colonne. La valeur par défaut est la table principale de l'entité. Cet attribut est utilisé lorsqu'un Entity Bean est *mappé* à plusieurs tables.
- `length` : précise la longueur que la base de données doit associer à un champ texte. La longueur par défaut est 255.
- `precision` : précise le nombre maximum de chiffre que la colonne peut contenir. La précision par défaut est définie par la base de données.
- `scale` : précise le nombre fixe de chiffres après le séparateur décimal (en général le point). Cet attribut n'est utilisable que pour les propriétés décimales (float, double ...). Le nombre de décimal par défaut est défini par la base de données.

```
@Column(updatable = true, name = "price", nullable = false,
  precision=5, scale=2)
public float getPrice() { return price; }
```

Les nombreux paramètres par défaut de l'API Persistence offrent un avantage certain au développeur. Celui-ci peut rapidement tester ses entités. Toutefois, il ne doit pas en rester là, mais utiliser les possibilités exposées ici pour optimiser le *mapping* entre ses entités et la base de données.

Objets incorporés

Lorsque sont définies les entités d'une application, on souhaite parfois regrouper un sous-ensemble de propriétés autour d'une classe. Par exemple, une entité `Person` possède une adresse `address` (avec les attributs : `street`, `zipCode`, `city`), de la même

façon une entité Order possède également une adresse (de livraison). Toutefois, dans un souci de séparation stricte, ces adresses doivent se trouver respectivement dans la table Person ou Order (suivant l'entité) et non dans une table Address annexe.

Une première solution est de définir toutes les composantes de l'adresse dans les classes Person et Order. Cependant, cette solution ne semble pas très orientée objet.

La meilleure solution est de créer une classe Address (qui n'est pas un Entity Bean). Puis de l'utiliser comme un type dans les classes d'entité Person et Order. C'est le principe même de l'agrégation en POO (Programmation orienté objet).

Cette dernière solution a été retenue dans la spécification EJB 3 et il est facile de l'implémenter. On appelle ces objets des *embedded objects* (objets incorporés). Ci-après, les trois classes de notre exemple :

```
@Embeddable
public class Address implements Serializable {
  @Column(length=30)
  private String address1;

  @Column(length=30)
  private String address2;

  @Column(length=12)
  private String zipCode;

  public Address() {}

  public String getAddress1() { return address1; }
  public void setAddress1(String address1) { this.address1 = address1; }

  public String getAddress2() { return address2; }
  public void setAddress2(String address2) { this.address2 = address2; }

  public String getZipCode() { return zipCode; }
  public void setZipCode(String zipCode) { this.zipCode = zipCode;}
}
```

La classe définissant le type personnalisé de propriété doit être annotée avec @Embeddable. Il est possible de définir des *mappings* par défaut pour chacune des propriétés incluses dans cette classe *via* les annotations de propriétés et/ou de colonnes.

```
@Entity
public class Person {
  @Id
       private String firstName;

  @Embedded
       private Address address;

  public Person() { }
  public Address getAddress() { return address; }
  public void setAddress(Address address) { this.address = address; }
}
```

Pour utiliser le type personnalisé, il faut définir une propriété avec ce type et l'annoter avec @Embedded. Cette annotation précise que les composants du type utilisé (ici Address) doivent être « incorporés » avec les autres propriétés de l'entité, dans la table associée à l'entité.

```
@Entity
@Table(name="ORDERS")
public class Order {
  private @Id int id;
  private @Temporal(TemporalType.DATE) Date createdDate;

  @Embedded

  @AttributeOverrides({
    @AttributeOverride(name="address1",
      column=@Column(name="shipppingAddress1")),
    @AttributeOverride(name="address2",
      column=@Column(name="shippingAddress2")),
    @AttributeOverride(name="zipCode", column=@Column(name="shippingZipCode"))
})
  private Address shippingAddress;

  @Embedded
  @AttributeOverrides({
    @AttributeOverride(name="address1",
      column=@Column(name = "invoicingAddress1")),
    @AttributeOverride(name="address2",
      column=@Column(name = "invoicingAddress2")),
    @AttributeOverride(name="zipCode",
      column=@Column(name = "invoicingZipCode")) })
  private Address invoicingAddress;

  public Order() { }

  public int getId() { return id; }

  public void setId(int id) { this.id = id; }

  public Date getCreatedDate() { return createdDate; }

  public void setCreatedDate(Date createdDate) {
    this.createdDate = createdDate;
  }

  public Address getInvoicingAddress() { return invoicingAddress; }

  public void setInvoicingAddress(Address invoicingAddress) {
    this.invoicingAddress = invoicingAddress;
  }

  public Address getShippingAddress() { return shippingAddress; }

  public void setShippingAddress(Address shippingAddress) {
    this.shippingAddress = shippingAddress;
  }
}
```

Il est parfois utile de surdéfinir les noms de colonnes utilisés par défaut dans un type personnalisé. Des problèmes apparaissent si vous utilisez plus d'une fois un même type personnalisé dans une même entité. C'est le cas de notre exemple, où une commande `Order` possède une adresse de livraison `shippingAddress` et une adresse de facturation `invoicingAddress` toutes deux de type `Address`. Il est donc obligatoire d'utiliser différents champs de la table pour chaque adresse.

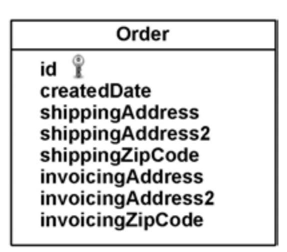


Figure 4.8 — Schéma de la table Order

L'annotation `@AttributeOverrides` pourra être ajoutée sur une propriété déjà annotée avec `@Embedded` pour effectuer cette surdéfinition. Celle-ci contiendra un ensemble d'annotations `@AttributeOverride` (sans « s ») dont l'attribut `name` permet de spécifier le nom de propriété sur laquelle porte la surdéfinition. De même, l'attribut `column` permet de préciser la nouvelle définition de la colonne *via* un objet `@Column`.

4.3.4 Identificateur unique (clé primaire)

Un Entity Bean doit posséder un champ dont la valeur est unique, dit identificateur unique ou clé primaire. Ce champ permet de différencier chaque instance de l'entité des autres. Cette clé primaire doit être définie une seule fois dans toute la hiérarchie de l'Entity Bean.

Il existe deux types d'identifiants : simple et composite. Nous expliquons, dans cette partie, l'utilisation de ces deux types, ainsi que leurs différences.

Identifiant simple

On parle d'identifiant simple lorsque celui-ci est composé par un unique champ dont le type est « simple ». Les types simples sont :

- les types primitifs (`int`, `float`, `char`...)
- les enveloppeurs (*wrapper*, en anglais. Par exemple `Integer`, `Float`, `Double`...)
- les types `String` ou `Date` (`java.util.Date` ou `java.sql.Date`).

> **Remarque :** en général, les décimaux (nombre à virgule) ne sont pas utilisés en tant que clé primaire. Les entités utilisant ce type pour la clé primaire risquent de ne pas être portables.

Pour spécifier au conteneur qu'un champ est une clé primaire, il faut annoter celui-ci avec `@Id`. Dans notre Entity Bean `User`, la clé primaire est assignée au champ `id`.

```
@Entity
public class User implements Serializable {
  private int id;

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO) // optionnel
  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }
...
}
```

L'exemple précédent utilise l'annotation `@GeneratedValue` qui permet d'indiquer au conteneur d'utiliser la meilleure solution pour la génération de la clé primaire. Il existe quatre stratégies de génération disponibles : `AUTO`, `IDENTITY`, `SEQUENCE` et `TABLE`. Celles-ci sont définies par l'énumération `javax.persistence.GenerationType`.

Le type `IDENTITY` indique au fournisseur de persistance d'assigner la valeur de la clé primaire en utilisant la colonne identité de la base de données. Sous MySQL, par exemple, la clé primaire auto-générée est marquée avec « AUTO_INCREMENT ».

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
public int getId() { ... }
```

Le type `SEQUENCE`, comme son nom l'indique, oblige le fournisseur de persistance à utiliser une séquence de la base de données. Celle-ci peut être déclarée au niveau de la classe ou au niveau du package grâce à l'annotation `@SequenceGenerator` et ses attributs :

- `name` (requis) : définit un nom unique pour la séquence qui peut être référencée par une ou plusieurs classes (suivant le niveau utilisé pour la déclaration de l'annotation).
- `sequenceName` (optionnel) : définit le nom de l'objet séquence de la base de données qui sera utilisé pour récupérer les valeurs des clés primaires liées.
- `initialValue` (optionnel) : définit la valeur à laquelle doit démarrer la séquence.

- `allocationSize` (optionnel) : définit le nombre utilisé pour l'incrémentation de la séquence lorsque le fournisseur de persistance y accède.

Attention : la valeur par défaut pour l'attribut `allocationSize` est 50.

```
@Entity
@SequenceGenerator (
  name="SEQ_USER",
  sequenceName="SEQ_USER"
)
public class User {
  private int id;

  @Id
  @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ_USER")
  public int getId() { return id; }
}
```

Ce type de génération est utile lorsque la base de données offre un système natif de séquence et qu'il est conseillé de l'utiliser par le fournisseur de celle-ci.

Le type `TABLE` indique au fournisseur de persistance d'utiliser une table annexe pour générer les clés primaires numériques. Ce cas d'utilisation est le plus rare.

```
@Entity
@TableGenerator (
  name="CONTACT_GEN",
  table="GENERATOR_TABLE",
  pkColumnName = "key",
  valueColumnName = "hi",
  pkColumnValue="id",
  allocationSize=25)
public class User {
  private int id;

  @Id
  @GeneratedValue(strategy=GenerationType.TABLE, generator="CONTACT_GEN")
  public int getId() { return id; }
}
```

L'annotation `@TableGenerator` permet de préciser les paramètres de création de la table annexe de génération. Voici le détail des attributs de celle-ci :

- `name` : définit un nom pour cette définition de table annexe.
- `table` : définit le nom de la table dans la base de données.
- `pkColumnName` : spécifie le nom de la colonne qui identifie la clé primaire spécifique pour laquelle la clé est générée.
- `valueColumnName` : spécifie le nom de la colonne qui contient le compteur de la clé primaire.
- `pkColumnValue` : spécifie la colonne de la clé primaire liée.

- `allocationSize` : définit le nombre d'incrémentations effectuées lorsque le fournisseur demande à la table une nouvelle valeur. Cela permet au fournisseur d'utiliser un système de cache afin de ne pas demander une nouvelle valeur à chaque demande d'un nouvel id.

Le type `AUTO` indique au fournisseur de persistance d'utiliser la meilleure stratégie (entre `IDENTITY`, `TABLE`, `SEQUENCE`) suivant la base de données utilisée. Le générateur `AUTO` est le type préféré pour avoir une application portable.

Même si l'incrémentation automatique de la clé primaire soulage le développeur, elle doit être utilisée avec parcimonie. En effet, il est préférable d'utiliser un champ de l'entité plutôt que d'en ajouter un, spécialement pour la clé primaire. Par exemple, l'entité `Compte` (`Account`, en anglais) peut contenir une propriété `numeroDeCompte` (`accountNumber`, en anglais) qui se veut unique par la logique bancaire. Cette propriété est alors la meilleure candidate pour la clé primaire.

Identifiant composite

Une clé primaire non « simple » (objet personnalisé…), pour être unique, doit rassembler une combinaison de propriétés. On parle alors de clé primaire composite ou identifiant composite.

La première étape pour utiliser une clé primaire composite est de créer une classe de clé primaire. Cette classe doit :

- Définir les propriétés devant être liées à l'identifiant unique.
- Avoir une visibilité `public`.
- Avoir un constructeur public sans argument.
- Implémenter l'interface `java.io.Serializable`.
- Surdéfinir les méthodes `equals()` et `hashCode()`. L'implémentation de ces deux méthodes doit correspondre à celle de la base de données. Cela signifie que vos méthodes et la base de données doivent retourner le même résultat (égalité ou inégalité) par rapport à la comparaison de deux clés primaires.
- Déclarer au conteneur une classe de clé primaire composite, *via* l'annotation `@Embeddable`.

Voici un exemple de clé primaire :

```
@Embeddable
public class ContactPK implements Serializable {
  private String firstName;
  private String lastName;

  // Constructeur public sans argument
  public ContactPK() {
  }

  public String getFirstName() {
    return firstName;
  }
```

```
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  @Override
  public boolean equals(Object obj) {
    // vérifie que l'objet passé en paramètre est de type ContactPK
    if(obj instanceof ContactPK) {
      ContactPK pk = (ContactPK) obj;
      // vérifie si les prénoms sont égaux
      if(this.getFirstName().equals(pk.getFirstName())) {
        // vérifie si les noms sont égaux
        if(this.getLastName().equals(pk.getLastName())) {
          return true;
        }
      }
    }
    return false;
  }

  @Override
  public int hashCode() {
    return (getFirstName() + getLastName()).hashCode();
  }
}
```

Il existe, ensuite deux façons d'utiliser cette classe au sein d'un Entity Bean. La première, est de définir une propriété avec le type de votre clé primaire. Cette propriété doit être annotée avec `@EmbeddedId` et non `@Id`.

```
@Entity
public class Contact {

  private ContactPK contactPK;

  private String address;

  protected Contact() {
  }

  public Contact(String firstName, String lastName) {
    ContactPK contactPK = new ContactPK();
    contactPK.setFirstName(firstName);
    contactPK.setLastName(lastName);
    setContactPK(contactPK);
  }
```

```
  @EmbeddedId
  public ContactPK getContactPK() {
    return contactPK;
  }

  public void setContactPK(ContactPK contactPK) {
    this.contactPK = contactPK;
  }

  public String getAddress() {
    return address;
  }

  public void setAddress(String address) {
    this.address = address;
  }
}
```

La seconde façon n'utilise pas explicitement la classe de la clé primaire mais y fait référence *via* l'annotation `@IdClass`. La classe de l'Entity Bean déclare explicitement les composantes de la clé primaire et chacune de celle-ci est annotée avec `@Id`.

```
@Entity
@IdClass(value=ContactPK.class)
public class Contact {

  private String firstName;

  private String lastName;

  private String address;

  public String getAddress() {
    return address;
  }

  @Id
  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  @Id
  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public void setAddress(String address) {
    this.address = address;
  }
}
```

Attention : cette seconde méthode impose que les noms des propriétés utilisées dans la classe de clé primaire et ceux de la classe de l'entité soient les mêmes. Dans notre exemple, nous retrouvons `firstName` et `lastName` dans chacune des deux classes.

Exceptée la syntaxe, aucune différence n'existe réellement entre ces deux méthodes. Le *mapping* avec la base de données est strictement le même. La première méthode semble plus optimale car il n'y a pas redondance d'information entre la classe de clé primaire et celle de l'entité mais vous oblige à travailler avec une clé primaire « complexe ». La deuxième méthode, quant à elle, est cependant plus intuitive et plus transparente. En effet, le développeur n'a pas réellement conscience de la clé primaire composite.

4.3.5 Les champs relationnels

Nous avons vu précédemment la définition de propriété persistante. Il s'avère qu'une entité ne travaille généralement pas seule mais qu'elle est reliée à d'autres entités. On parle alors de relations entre entités. Les types de relations sont les mêmes qu'en EJB 2. Pour définir un champ relationnel au sein d'une entité, il suffit de créer une propriété dont le type est une entité (cardinalité 1) ou un ensemble d'entité (cardinalité *n*). Une relation est dite unidirectionnelle si une seule partie connaît la relation. À l'opposé, elle est qualifiée de bidirectionnelle si les deux parties la connaissent.

Pour les relations dont un côté, au moins, est multi-valué (« One to Many », « Many to One », « Many to Many »), les types de conteneurs disponibles sont : `java.util.Collection`, `java.util.Set`, `java.util.List`, et `java.util.Map`.

Ce sont ces relations entre entités EJB 3 que nous détaillons dans les parties qui suivent.

Un à Un (One to One)

Une relation « One to One » est utilisée pour lier deux entités uniques indissociables. Par exemple, un corps n'a qu'un seul cœur, ou une personne n'a qu'une seule carte d'identité. Nous supposerons, dans notre exemple, qu'un utilisateur `User` n'a qu'un seul compte `AccountInfo`.

Pour associer deux entités avec ce type de relation, il faut utiliser l'annotation `@OneToOne`. Celle-ci reprend les attributs de `@Basic`, vu précédemment, et propose d'autres attributs optionnels :

- `cascade` : spécifie les opérations à effectuer en cascade.
- `mappedBy` : spécifie le champ propriétaire de la relation dans le cas d'une relation bidirectionnelle.

- `targetEntity` : spécifie la classe de l'entité cible. Cet attribut est peu utilisé car l'annotation utilise automatiquement le type de la propriété.

Ce type de relation peut être *mappé* de trois manières dans la base de données.

La première solution consiste à utiliser les mêmes valeurs pour les clés primaires des deux entités. Il faut alors préciser ce choix *via* l'annotation `@PrimaryKeyJoinColumn` (jointure par clé primaire). Voici l'exemple de la relation unidirectionnelle entre `User` et `AccountInfo` avec cette méthode.

```
@Entity
public class AccountInfo {
//...
  private int id;
  private String cardNumber;
  private double amount;
  private String accountId;

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  public int getId() { return id; }
  public void setId(int id) { this.id = id; }

  public String getCardNumber() { return cardNumber; }
  public void setCardNumber(String cardNumber) {
    this.cardNumber = cardNumber;
  }
//...
}
```

```
@Entity
public class User {
//...
  @OneToOne
  @PrimaryKeyJoinColumn
  public AccountInfo getAccountInfo() {
    return accountInfo;
  }
//...
}
```

La deuxième solution consiste à utiliser une clé étrangère d'un côté de la relation. Toutefois, il faut noter que la colonne de cette clé doit être marquée comme unique afin de simuler correctement la relation « One to One ». L'annotation à utiliser est `@JoinColumn`. Elle permet de paramétrer la colonne de jointure à utiliser. L'exemple montre la relation bidirectionnelle.

```
@Entity
public class User {
//...
```

```
  @OneToOne
  @JoinColumn(name="account_id", referencedColumnName="id")
  public AccountInfo getAccountInfo() {
    return accountInfo;
  }
//...
}
```

L'annotation @JoinColumn ressemble à @Column mais a un attribut supplémentaire optionnel : referencedColumnName. Celui-ci permet de spécifier le nom de la colonne référencée par la clé étrangère de l'association.

```
@Entity
public class AccountInfo {
//...
  private int id;
  private User user;

  @OneToOne(mappedBy = "accountInfo")
  public User getUser() {
    return user;
  }
//...
}
```

L'attribut mappedBy déclare que le côté propriétaire est celui détenant la propriété accountInfo. C'est donc, ici, l'entité User qui détient la relation et a donc le pouvoir de lier un utilisateur à un compte (l'inverse étant impossible).

La dernière solution consiste à utiliser une table d'association des liens entre les deux entités. Toutefois, la multiplicité « One to One » est respectée si et seulement si une contrainte unique est définie sur chaque clé étrangère. Même si ce cas est plus rare, il est possible de la retrouver dans un système existant que l'on souhaite faire évoluer.

```
@Entity
public class User {
//...
  @OneToOne
  @JoinTable(name = "UserAccountInfo"
    joinColumns = @JoinColumn(name="user_fk", unique=true),
    inverseJoinColumns = @JoinColumns(name="accountinfo_fk", unique=true)
  )
  public AccountInfo getAccountInfo() { ... }
}
```

```
@Entity
public class AccountInfo {
//...
  @OneToOne(mappedBy = "accountInfo")
  public User getUser() { ... }
}
```

Cette technique oblige à écrire plus de lignes, sans pour autant augmenter les performances, voir l'inverse. C'est cependant un cas non négligeable lors de l'utilisation d'une source de données déjà existante. L'annotation `@JoinTable` permet de configurer la table de jointure pour la relation. Voici une description des attributs de celle-ci :

- `name` : spécifie le nom de la table de jointure.
- `catalog` : spécifie le catalogue de la table de jointure.
- `schema` : spécifie le schéma de la table de jointure.
- `joinColumns` : spécifie les colonnes (ensemble de `@JoinColumn`) de la table de jointure qui référencent la ou les clés primaires de l'entité propriétaire de la relation (côté propriétaire).
- `inverseJoinColumns` : spécifie les colonnes (ensemble de `@JoinColumn`) de la table de jointure qui référencent la ou les clé(s) primaire(s) de l'entité non propriétaire (côté inverse).
- `uniqueConstraints` : spécifie les contraintes uniques à placer dans la table. Cet attribut est utilisé seulement si la génération de la table est activée.

En l'absence de tout paramétrage de l'annotation `@OneToOne`, la liaison entre les deux entités se fera avec une colonne de jointure (deuxième solution) dans l'entité propriétaire. Le nom de celle-ci sera le résultat de la concaténation du nom de la propriété relationnelle avec le nom de la clé primaire de l'autre entité. Concrètement, dans l'exemple utilisé précédemment, l'entité propriétaire `User` définie la propriété persistante nommée `accountInfo` ; la clé primaire de `AccountInfo` étant `id`, le nom serait `accountInfo_id` .

Un à Plusieurs (One To Many) et Plusieurs à Un (Many To One)

Une relation « One To Many », et respectivement « Many To One », est utilisée pour lier à une unique instance d'une entité A, un groupe d'instances d'une entité B. Par exemple, une personne possède plusieurs comptes bancaires, mais un compte bancaire n'appartient qu'à une seule personne. Dans nos exemples suivants, un utilisateur peut avoir plusieurs portefeuilles d'actions, mais un portefeuille n'est lié qu'à un seul utilisateur.

Une association « Many To One » est définie sur une propriété avec l'annotation `@ManyToOne`. Dans le cas d'une relation bidirectionnelle, l'autre côté doit utiliser l'annotation `@OneToMany`. Les attributs de ces deux annotations correspondent à ceux de l'annotation `@OneToOne`.

```
@Entity
public class Portfolio {
//...
  private User user;


  @ManyToOne
  @JoinColumn(name = "user_fk")
```

```
  public User getUser() {
    return user;
  }

  public void setUser(User user) {
    this.user = user;
  }
}
```

```
@Entity
public class User implements Serializable {
//...
  private Collection<Portfolio> portfolios;

  @OneToMany(mappedBy = "user")
  public Collection<Portfolio> getPortfolios() {
    return portfolios;
  }

  public void setPortfolios(Collection<Portfolio> portfolios) {
    this.portfolios = portfolios;
  }
}
```

L'entité `Portfolio`, étant la part de la relation ayant une cardinalité de 1, définit une propriété `User user`. À l'opposé, l'entité `User`, part de la relation à la cardinalité « n », déclare une propriété multi-valuée `Collection<Portfolio> portfolios`.

Remarquez l'utilisation des génériques qui évitent de définir l'entité cible dans la définition de notre relation. Si vous ne souhaitez pas les utiliser, alors il vous faudra spécifier la cible de votre relation *via* l'attribut `targetEntity` de l'annotation `@OneToMany`.

Remarque : « Many To One » signifie « plusieurs **vers un** ». Elle est donc utilisée sur une propriété de **cardinalité 1**. « One To Many » signifie « un **vers plusieurs** ». Elle est donc appliquée sur une propriété de **cardinalité n** (type Collection, par exemple).

De la même façon qu'avec une relation « One to One », il est possible d'utiliser une table d'association. La manière de procéder ayant déjà été expliquée précédemment, nous ne reviendrons pas dessus.

Plusieurs à Plusieurs (Many To Many)

Le dernier type de relation disponible est « Many to Many ». Il peut être utilisé pour lier des instances de deux entités entre elles. Par exemple, entre des articles et des catégories. Un article peut être associé à plusieurs catégories (cardinalité *n*) et une catégorie peut regrouper plusieurs articles (cardinalité *m*).

Pour cela, il suffit d'utiliser des propriétés multi-valuées de chaque côté de la relation (si celle-ci est bidirectionnelle) et de les annoter avec `@ManyToMany`. En terme relationnel, cette relation impose l'utilisation d'une table d'association.

```
@Entity
public class User {
//...

  private Collection<Hobby> hobbies;

  @ManyToMany
  @JoinTable(name = "USER_HOBBIES",
    joinColumns = @JoinColumn(name = "user_id", referencedColumnName = "id"),
    inverseJoinColumns = @JoinColumn(name = "hobby_id",
      referencedColumnName = "id"))
  public Collection<Hobby> getHobbies() {
    return hobbies;
  }

  public void setHobbies(Collection<Hobby> hobbies) {
    this.hobbies = hobbies;
  }
}
```

```
@Entity
public class Hobby {
//...
  private Collection<User> users;

  @ManyToMany(mappedBy="hobbies")
  public Collection<User> getUsers() {
    return users;
  }

  public void setUsers(Collection<User> users) {
    this.users = users;
  }
}
```

Ici, chaque utilisateur `User` possède un ensemble de hobbies `Hobby`. Et inversement, chaque hobby `Hobby` peut être lié à plusieurs utilisateurs `User`.

Opérations en cascade

Les annotations `@OneToOne`, `@OneToMany`, `@ManyToOne` et `@ManyToMany` possèdent l'attribut `cascade`. Celui-ci spécifie les opérations à effectuer en cascade. La cascade signifie qu'une opération appliquée à une entité est propagée aux relations de celle-ci (voir chapitre 6, pour les différentes notions le concernant). Par exemple, lorsqu'un utilisateur est supprimé, son compte l'est également.

Il existe quatre opérations possibles sur les entités : ajout, modification, suppression, rechargement. Ces opérations sont regroupées dans l'énumération `CascadeType` :

- `CascadeType.PERSIST` : automatise l'enregistrement des entités liées à l'association marquée lors de l'enregistrement de l'entité propriétaire (méthode `persist()`).
- `CascadeType.MERGE` : automatise l'enregistrement des modifications des entités liées à l'association marquée, lors de l'enregistrement des modifications de l'entité propriétaire (méthode `merge()`).
- `CascadeType.REMOVE` : automatise la suppression des entités liées à l'association marquée, lors de la suppression de l'entité propriétaire (méthode `remove()`).
- `CascadeType.REFRESH` : automatise le rechargement (côté base de données) des entités liées à l'association marquée, lors du rechargement de l'entité propriétaire (méthode `refresh()`).
- `CascadeType.ALL` : cumule les quatre types de cascade.

Attention : selon la spécification, le type `CascadeType.REMOVE` ne peut être appliqué qu'aux associations « One to One » ou « One to Many ». L'utilisation de ce type pour d'autres associations n'est pas portable.

Par exemple, sur la relation entre `User` et `Portfolio`, il est logique d'enregistrer ou supprimer les portefeuilles lorsque l'utilisateur est respectivement enregistré ou supprimé. Voici le code correspondant :

```
@OneToMany(cascade = { CascadeType.REMOVE, CascadeType.PERSIST },

  mappedBy = "user")
public Collection<Portfolio> getPortfolios() {
  return portfolios;
}
```

De la même façon, la relation « One to One » entre `User` et `AccountInfo` oblige à sauvegarder, mettre à jour et détruire `AccountInfo` lorsque ces opérations sont effectuées sur l'instance du `User` correspondant. Voici le code correspondant :

```
@OneToOne(cascade = CascadeType.ALL)
@PrimaryKeyJoinColumn
public AccountInfo getAccountInfo() {
 return accountInfo;
}
```

L'utilisation du mécanisme de cascade est une réelle simplification pour le développeur. En effet, il n'a plus à gérer les boucles de suppression, modification… Toutefois, cet outil doit être utilisé judicieusement et avec parcimonie. En effet, une utilisation trop importante de ce mécanisme peut très vite nuire aux performances de l'application.

Un Many to Many ou deux One to Many ?

Une relation « Many to Many » n'est ni plus ni moins que le regroupement de deux relations « One to Many ». En effet, la relation « Many to Many » utilise une table de jointure simple contenant les clés primaires des deux côtés de la relation.

Qu'en est-il si l'on souhaite ajouter des propriétés à cette relation ? Par exemple, lorsqu'une commande regroupe des produits, ceux-ci peuvent aussi être utilisés dans plusieurs commandes. Nous nous retrouvons alors dans le cas d'une relation « Many to Many ». Cependant, il est généralement utile d'ajouter une propriété concernant, par exemple, la quantité du produit désiré.

L'utilisation de l'annotation `@ManyToMany` ne permet pas l'insertion de propriétés annexes à la relation entre les deux objets. Il est alors nécessaire d'utiliser une double liaison « One to Many » et un Entity Bean intermédiaire. Dans notre exemple, nous avons les Entity Beans suivants : `Order`, `Product` et `OrderLine`, représentant `respectivement` une commande, un produit et les lignes des commandes. Voici les codes respectifs :

```
@Entity
@Table(name="ORDERS")
public class Order {

  private int id;

  private Date orderDate;

  private Collection<OrderLine> orderLines;

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public Date getOrderDate() {
    return orderDate;
  }

  public void setOrderDate(Date orderDate) {
    this.orderDate = orderDate;
  }

  @OneToMany(mappedBy = "order", cascade = {CascadeType.REMOVE})
  public Collection<OrderLine> getOrderLines() { return orderLines; }

  public void setOrderLines(Collection<OrderLine> ol) {this.orderLines = ol;}
}
```

```
@Entity
public class Product {

  private int id;

  private String name;

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

Les classes `Order` et `Product` sont des Entity Beans comme nous avons pu les décrire dans les premières parties.

L'Entity Bean `OrderLine` est le point central de la relation. Il détient les deux propriétés `product` et `order` respectivement liées aux Entity Beans `Product` et `Order`. De plus, même si une clé primaire simple auto-générée simplifierait le travail du développeur, il est plus judicieux de travailler avec une clé primaire composite. En effet, le regroupement des clés primaires de `Product` et `Order` est un bon candidat pour une clé primaire, étant donné qu'une commande ne peut avoir deux fois le même produit (dans ce cas-là il suffit d'incrémenter la quantité). Nous devons alors créer une classe `OrderLinePk` pour définir cette clé primaire composite. Celle-ci contient deux propriétés `orderId` et `productId` représentant respectivement les id de `Order` et `Product`.

```
@Embeddable
public class OrderLinePk implements Serializable {

  private static final long serialVersionUID = 1L;

  private int orderId;

  private int productId;

  protected OrderLinePk() {
  }

  public OrderLinePk(int orderId, int productId) {
    this.orderId = orderId;
    this.productId = productId;
  }
```

```
  public int getOrderId() {
    return orderId;
  }

  public void setOrderId(int orderId) {
    this.orderId = orderId;
  }

  public int getProductId() {
    return productId;
  }

  public void setProductId(int productId) {
    this.productId = productId;
  }

  public int hashCode() { return orderId ^ productId; }

  public boolean equals(Object that) {
    return (that instanceOf OrderLinkPk && orderId == productId);
  }
}
```

```
@Entity
@IdClass(OrderLinePk.class)
public class OrderLine {

  private Product product;

  private Order order;

  private int quantity;

  public OrderLine() {
  }

  public int getQuantity() {
    return quantity;
  }

  public void setQuantity(int quantity) {
    this.quantity = quantity;
  }

  @ManyToOne
  @JoinColumn(name="orderId",
              optional=false,
              insertable=false,
              updatable=false)
  public Order getOrder() {
    return order;
  }
```

```
  public void setOrder(Order order) {
    this.order = order;
  }

  @ManyToOne
  @JoinColumn(name="productId",
              optional=false,
              insertable=false,
              updatable=false)
  public Product getProduct() {
    return product;
  }

  public void setProduct(Product product) {
    this.product = product;
  }

  @Id
  public int getProductId() {
    return getProduct().getId();
  }

  @Id
  public int getOrderId() {
    return getOrder().getId();
  }

  public void setOrderId(int orderId) {
    getOrder().setId(orderId);
  }

  public void setProductId(int productId) {
    getProduct().setId(productId);
  }
}
```

Nous utilisons l'annotation `@IdClass` afin de spécifier la classe de la clé primaire utilisée. Les getters `getProductId()` et `getOrderId()` surdéfinissent, en quelque sorte, ceux de la classe `OrderLinePk`. Toutefois, ils retournent respectivement la clé primaire de la commande liée `Order` et celle du produit lié `Product`.

Cela soulève un problème : les colonnes « orderId » et « productId » sont déjà utilisées pour référencer dans les relations « Many to One ». Elles sont donc utilisées à la fois comme clé primaire et clé étrangère ! Le fournisseur d'entités se heurte donc à un problème : assigner la clé primaire et les clés étrangères. Pour spécifier qu'on ne souhaite pas assigner automatiquement les clés étrangères, il est nécessaire d'affecter la valeur `false` aux attributs `insertable` et `updatable` de l'annotation `@JoinColumn`.

4.3.6 L'héritage

L'héritage est un nouvel aspect important des Entity Beans qui n'existait pas avec les EJB 2. En effet, il s'agit d'une notion essentielle de la programmation objet qui se retrouve dans la quasi-totalité des applications d'aujourd'hui. Toutefois, la représen-

tation relationnelle d'un héritage d'objets n'est pas une chose simple et plusieurs solutions sont possibles.

Voici les trois types de *mappings* relationnels possibles :

- Une table unique par hiérarchie de classe.
- Une table par classe concrète.
- Une séparation des champs spécifiques d'une classe fille dans une table séparée de la table parente. Une jonction est alors faite pour instancier la classe fille.

Les exemples de cette partie utiliseront deux Entity Beans : `Bond` et `Stock` qui héritent tous deux de l'Entity Bean `FinancialProduct` (classe abstraite).

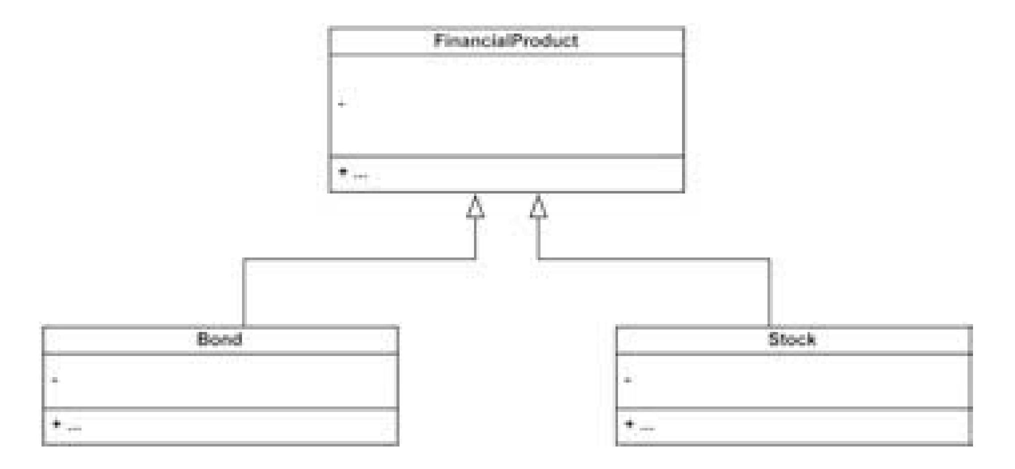


Figure 4.9 — Héritage de `Bond` et `Stock` vers `FinancialProduct`

> **Attention :** une seule clé primaire doit être définie dans une hiérarchie. Ici, la clé primaire se trouve dans la classe racine `FinancialProduct`.

Nous allons décrire les possibilités de chacune de ces configurations, ainsi que leurs avantages et inconvénients. Le but est de vous donner les éléments vous permettant de faire le meilleur choix selon les différentes situations.

Pour chaque cas, nous étudierons les annotations utilisées et la structure relationnelle générée.

Une table unique

Dans cette stratégie, toutes les classes de la hiérarchie sont *mappées* dans une même et unique table. Le type d'héritage utilisé est spécifié au niveau de l'Entity Bean racine par l'annotation `@Inheritance`.

```
@Entity
@Inheritance (strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="financialproduct_type",
                     discriminatorType=DiscriminatorType.STRING,
                     length=10)
public abstract class FinancialProduct {
  private int id;
  //...
  @Id
  @GeneratedValue(strategy=GeneratorType.AUTO)
  public int getId(){
    return id;
  }
  //...
}
```

Le seul attribut de cette annotation est `strategy`. Il permet de spécifier le type de stratégie à utiliser *via* l'énumération `InheritanceType`. Pour définir un héritage utilisant la stratégie « table unique », c'est la valeur `InheritanceType .SINGLE_TABLE` qui sera utilisée.

Toutefois, cette stratégie requiert une colonne permettant de différencier les types d'entité de la hiérarchie. Il faut utiliser l'annotation `@DiscriminatorColumn` pour préciser les détails de cette colonne. Les attributs de cette annotation sont :

- `name` : nom de la colonne de discrimination.
- `discriminatorType` : classe du discriminateur à utiliser défini *via* l'énumération `DiscriminatorType`. Celle-ci possède les attributs suivants :
  – `CHAR`
  – `INTEGER`
  – `STRING`
- `length` : taille de la colonne pour les discriminateurs à base de chaîne de caractères.
- `columnDefinition` : fragment SQL à utiliser pour la déclaration de la colonne (utilisé lors de la génération des tables par le conteneur).

Les classes filles concrètes peuvent préciser la valeur discriminatoire grâce à l'annotation `@DiscriminatorValue`. Par défaut, la valeur d'un discriminateur de type `STRING` contient le nom de la classe de l'Entity Bean. Ainsi, le contenu de la colonne discriminatoire contient « Bond » si l'EntityBean stocké est de type `Bond`, et « Stock » si son type est `Stock`. Pour les autres types de discriminateur, le fournisseur utilise une méthode spécifique pour générer une valeur automatiquement pour l'Entity Bean.

```
@Entity
@DiscriminatorValue("BOND") // valeur par défaut
public class Bond extends FinancialProduct {
  private double rate;       // taux de rendement de l'obligation
  private int monthDuration; // nombre de mois que dure l'obligation
   //...
   @Basic
```

```
    public double getRate(){
      return rate;
    }
    @Basic
    public int getMonthDuration(){
      return monthDuration;
    }
    //...
}
```

```
@Entity
@DiscriminatorValue("STOCKOPTION")
public class Stock extends FinancialProduct {
...
}
```

La figure 4.10 présente le résultat au niveau de la base de données (une seule table pour deux entités).

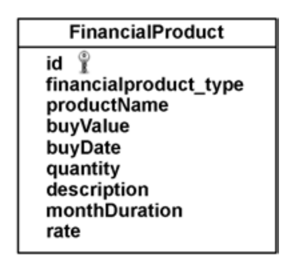


Figure 4.10 — Modèle conceptuel des données (MCD) de la table FinancialProduct

Cette stratégie fournit un gain de performance important, car aucune jointure n'est réalisée. Cependant, les données utilisent plus de place car l'ensemble des colonnes n'est pas toujours utilisé (suivant le type de données enregistrées). Ici, `rate` a une valeur « null » lorsque l'enregistrement correspond à un Entity Bean `Stock`.

Une table par classe concrète

La seconde stratégie présentée ici est « une table par classe concrète ». Dans ce cas, chaque classe Entity Bean concrète est liée à sa propre table. Cela signifie que toutes les propriétés de la classe (incluant les propriétés héritées) sont incluses dans la table liée à cette entité.

Concrètement, sur notre exemple, cela signifie que la table « FinancialProduct » n'est pas créée ni utilisée. Les Entity Beans `Stock` et `Bond` qui héritent de la classe `FinancialProduct` sont *mappés* respectivement sur les tables « Stock » et « Bond » ; chacune de ces tables intégrant les propriétés définies dans `FinancialProduct`.

Pour spécifier cette stratégie, il faut spécifier la stratégie définie par `InheritanceType.TABLE_PER_CLASS` à l'annotation `@Inheritance`.

```
@Entity
@Inheritance (strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class FinancialProduct {
  //...
}
```

Les classes filles `Bond` et `Stock` ont simplement à hériter de `FinancialProduct`.

```
@Entity
public class Bond extends FinancialProduct {
  //...
}
```

```
@Entity
public class Stock extends FinancialProduct {
  //...
}
```

La figure 4.11 présente le résultat en base de données.

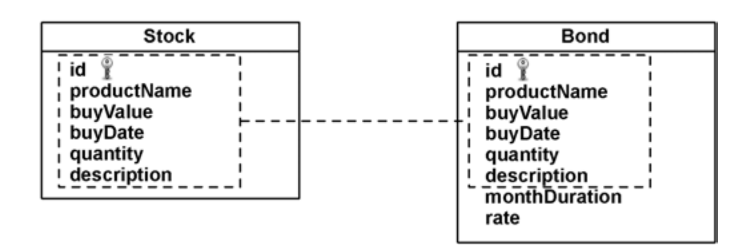


Figure 4.11 — MCD des tables « Stock » et « Bond »

L'avantage majeur de cette configuration se situe lors de l'insertion car l'ajout n'est fait que dans une seule table. De même, la sélection d'un type concret (ici `Bond` ou `Stock`) est optimisée. En effet, l'ordre SELECT porte sur une seule table.

L'inconvénient principal est la lourdeur des requêtes polymorphiques (sélection sur l'ensemble des `FinancialProduct`, par exemple). En effet, la base de données doit utiliser l'instruction SQL « UNION » afin de rassembler les enregistrements situés dans les deux tables.

L'autre inconvénient est la duplication des colonnes dans chacune des tables des Entity Bean ; les DBA (*DataBase Administrator*) n'apprécient pas toujours ce genre de pratique.

Tables jointes

Dans cette dernière stratégie, la classe racine des entités est représentée par une table. Chaque classe fille est liée à sa propre table séparée contenant les propriétés spécifiques de celle-là.

La liaison entre les tables « filles » et la table racine se fait *via* les clés primaires. En effet, la clé primaire de la classe fille est liée à celle de la classe parente. Ainsi, un

enregistrement de la table « Stock » ou « Bond » dont la valeur de la clé primaire est 15, est lié à un enregistrement de la table « FinancialProduct » ayant la valeur 15 pour la clé primaire.

L'héritage est ici déclaré avec la stratégie `InheritanceType.JOINED`.

```
@Entity
@Inheritance (strategy=InheritanceType.JOINED)
public class FinancialProduct {
 //...
}
```

Les classes fille `Bond` et `Stock` n'ont pas besoin d'autre annotation que `@Entity` :

```
@Entity
public class Bond extends FinancialProduct {
 //...
}
```

```
@Entity
public class Stock extends FinancialProduct {
  //...
}
```

La figure 4.12 présente le résultat au niveau de la base de données (optimisation de la place prise par les données).

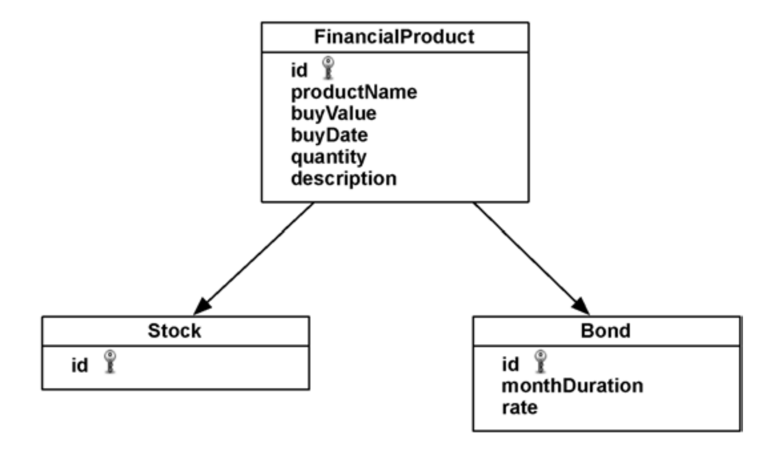


Figure 4.12 — MCD représentant le *mapping* de l'Entity Bean `FinancialProduct`

L'avantage de cette stratégie est d'avoir un modèle relationnel clair. C'est en quelque sorte le modèle idéal pour le DBA (pas de perte de mémoire car tous les champs sont utilisés...). Mais c'est aussi de fournir un bon support de la polymorphie. Ainsi, lorsque le client souhaite récupérer tous les Entity Beans de type `Finan-`

cialProduct, la requête SQL générée se traduira par un simple « SELECT * FROM FinancialProduct … ».

Cependant, l'inconvénient est qu'elle requiert l'utilisation de plusieurs jointures entre les tables lors de la récupération de données. Dans le cas de hiérarchies importantes (grande profondeur de l'héritage) cela peut entraîner de mauvaises performances.

Récapitulatif des méthodes de mapping de l'héritage

Le tableau 4.2 récapitule les avantages et les inconvénients des différentes stratégies.

Tableau 4.2 — Avantages et inconvénients des stratégies de *mapping*

Stratégie	SINGLE_TABLE	TABLE_PER_CLASS	JOINED
Avantages	Aucune jointure, donc très performant	Performant en insertion	Intégration des données proche du modèle objet
Inconvénients	Organisation des données non optimale	Polymorphisme lourd à gérer	Utilisation intensive des jointures, donc baisse de performance

L'héritage pouvant être très gourmand en ressources et performances, une autre solution peut être mise en place : l'utilisation de relations « One to One » ou « One to Many » avec le système de *lazy loading* (voir paragraphe 6.6.4).

En résumé

L'évolution des Entity Beans est sans doute la plus remarquable, celle-ci permettant de combler un manque crucial qui nuisait amplement aux EJB 2.

Dans la même logique que les Session Beans, les interfaces EJB 2 des Entity Beans disparaissent pour laisser place à une façon de penser plus intuitive.

De même, il est désormais possible d'utiliser un système d'héritage simple, qui faisait cruellement défaut aux EJB 2.

Désormais complet, le système de persistance EJB 3 tend, aujourd'hui, à devenir un standard en la matière.

5

Les Message Driven Beans

Objectif

Au sein d'une application entreprise de grande ampleur, il peut être intéressant de faire communiquer entre elles les différentes « sous-applications » clientes et serveurs. Par communication, il faut comprendre un envoi de données directement interprétables et utilisables par les autres applications.

Ce sont les Message Driven Beans qui permettent de traiter les messages venant d'autres applications.

Nous présenterons dans un premier temps l'API JMS, qui permet la connexion avec un système de messagerie inter-applications. Nous étudierons ensuite le développement de ce type de composant, ainsi que le rôle joué par le conteneur dans les versions EJB 2 et 3.

5.1 INTRODUCTION

Le concept de MDB (*Message Driven Bean*) a été introduit avec les EJB 2.0 afin de traiter les messages venant d'un fournisseur JMS (*Java Message Service*). Depuis la version 2.1, les MDB supportent n'importe quel système de messagerie et sont donc indépendants de JMS. La spécification EJB 3.0 n'offre pas réellement de nouvelles fonctionnalités, mais simplifie grandement le développement de ces composants.

Tous les serveurs d'applications compatibles EJB 3.0 doivent supporter JMS. Beaucoup d'éditeurs fournissent leur propre implémentation JMS, d'autres fournissent les supports pour interroger d'autres implémentations JMS. Dans tous les cas, un fournisseur JMS est inévitable pour utiliser les MDB.

Nous préciserons les détails essentiels de JMS pour son utilisation au sein des MDB. En effet, il nous est impossible d'expliquer l'ensemble de cette API qui couvrirait un ouvrage à elle seule.

5.2 JAVA MESSAGE SERVICE

5.2.1 Qu'est-ce que JMS ?

JMS est l'API utilisée pour l'accès à un système de messagerie d'entreprise. C'est la solution Java EE au concept du MOM (*Message Oriented Middleware*).

Ce système permet l'échange de messages entre différentes applications distantes. Les serveurs de messagerie sont souvent méconnus (contrairement aux serveurs de base de données), bien qu'ils soient nombreux. On retrouve MQSeries d'IBM, JBoss Messaging (anciennement JBoss MQ), One Message Queue de Sun Microsystem… Les applications utilisant JMS sont indépendantes du type de serveur auquel elles se connectent du moment qu'elles utilisent l'API JMS.

Les applications utilisent généralement JMS dans les architectures de type B2B (*Business to Business*). En effet, cette API permet d'interconnecter n'importe quel système utilisant le principe de messagerie où l'envoi et la réception de message sont asynchrones. Cela signifie que les applications communiquant *via* JMS peuvent ne pas s'exécuter en même temps, sur le même principe que le système de courrier électronique (*email*). Lorsque l'expéditeur envoie une requête (*via* un email), il ne reçoit pas directement la réponse. Il se peut qu'il ne reçoive jamais de réponse, ou seulement un accusé de réception.

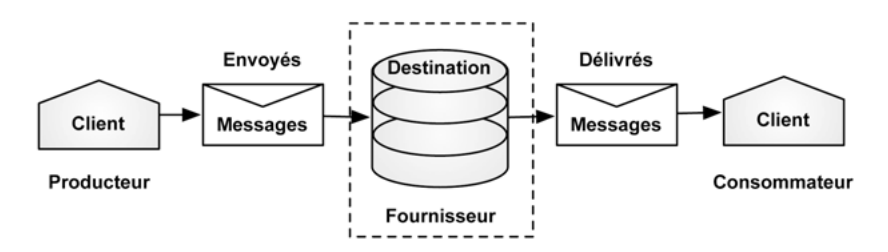


Figure 5.1 — Architecture JMS

L'architecture JMS (fig. 5.1) est composée de différents éléments :

- *Un fournisseur* : c'est l'élément qui a la charge de la livraison des messages entre les différents intervenants. Il s'occupe de traiter les envois et de faire en sorte qu'ils soient bien reçus.
- *Un client* : c'est une application ou composant d'application intervenant lors des échanges. Il envoie ou reçoit les messages.

- *Un message* : c'est, comme son nom l'indique, l'élément qui va transiter *via* une communication entre les clients. Un fournisseur sert toujours d'intermédiaire ; on ne les envoie donc pas directement d'un client à un autre.
- *Les destinations* : ce sont des objets configurés au niveau du fournisseur qui sont à disposition des clients et qui seront utilisés par ces derniers pour l'envoi et la réception des messages. Pour schématiser, on peut dire qu'il s'agit de « boîtes à lettres » dans lesquelles sont placés les messages en attendant qu'un client vienne les réclamer.

Un point important dans cette architecture est qu'elle permet une communication faiblement couplée entre les clients : un client ne se préoccupe pas de l'identité de son ou ses correspondants ni de leur éventuel état. De plus, ce système peut travailler en environnement hétérogène (application C++, Java...).

5.2.2 Les composants JMS

ConnectionFactory et Destination

Pour travailler avec JMS, la première étape est de se connecter au fournisseur JMS. Pour cela, il faut récupérer un objet `ConnectionFactory` *via* JNDI qui rend la connexion possible avec le fournisseur. Cet objet peut être assimilé à une `DataSource` (en JDBC). En effet, de la même façon qu'une `DataSource` fournit une connexion JDBC, une `ConnectionFactory` fournit une connexion JMS au service de routage de message.

Voici un exemple détaillant la récupération d'une `ConnectionFactory` et une `Destination` :

```
Context jndiContext = new InitialContext();
ConnectionFactory connectionFactory = (ConnectionFactory)
        jndiContext.lookup("ConnectionFactory");
Destination destination = (Destination) jndiContext.lookup("queue/MyQueue");
```

L'autre élément à récupérer est la destination. Elle représente l'endroit sur lequel l'application souhaite travailler (envoi ou réception de message). Nous verrons dans la partie suivante qu'il existe deux types de destinations : les *topics* et les *queues*.

Connection et Session

L'objet `ConnectionFactory` permet de créer une connexion avec le fournisseur JMS. Une fois la connexion créée, elle est utilisée pour créer une session.

```
Connection cnx = connectionFactory.createConnection();
Session session = cnx.createSession(true, 0);
```

La session sert à grouper les opérations d'envoi et de réception des messages. Dans la majorité des cas, une unique session est suffisante. La création de plusieurs sessions est utile seulement dans le cas d'application « multi-thread » qui produisent et reçoivent des messages en même temps. En effet, l'objet `Session` est « thread-

safe », c'est-à-dire que ses méthodes n'autorisent pas l'accès concurrent. Généralement, le thread qui crée l'objet `Session` utilise le producteur et le consommateur de cette session.

La méthode `createSession()` prend deux paramètres :

```
createSession (boolean transacted, int acknowledge)
```

Toutefois, la spécification indique que la valeur de ces arguments est ignorée au sein du conteneur EJB. En effet, celui-ci gère les transactions et les accusés de réception en fonction des paramètres de déploiement.

Attention : certains fournisseurs n'adhèrent pas totalement à la spécification et n'ignorent pas les paramètres.

Les bonnes pratiques de développement incitent à fermer les connexions une fois le travail terminé.

```
Connection cnx = connectionFactory.createConnection();
...
cnx.close();
```

MessageProducer et MessageConsumer

La dernière étape est l'envoi et la réception de messages. Pour cela, il faut créer des objets de type `MessageProducer` et `MessageConsumer` respectivement pour envoyer et recevoir des messages.

```
MessageProducer producer = session.createProducer(destination);
MessageConsumer consumer = session.createConsumer(destination);
```

Chacune des méthodes prend en paramètre la destination sur laquelle l'objet est connecté.

Type de Message

Dans JMS, un message est un objet Java constitué d'un en-tête ainsi que d'un corps. L'en-tête se compose des informations de destination, d'expiration, de priorité… Le corps du message contient des données pouvant être de différents types :

- Texte avec `TextMessage`.
- Objet sérialisé avec `ObjectMessage`.
- Map avec `MapMessage`.

Chacun de ces types de messages hérite de `javax.jms.Message`.

Prenons le cas d'un `MapMessage`, où un message demande l'achat de 250 actions de l'entreprise « JavaCorp ». Ces informations seraient transmises dans un objet de type `MapMessage :`

```
...
MapMessage mapMsg = session.createMapMessage() ;
mapMsg.setInt("action", StockAction.BUY_STOCK) ;
mapMsg.setString("company","JavaCorp") ;
mapMsg.setInt("number",250);
producer.send(mapMsg);
```

Une autre solution consiste à envoyer directement un objet sérialisable avec `ObjectMessage` :

```
StockAction stock = new StockAction();
stock.setAction(StockAction.BUY_STOCK);
stock.setCompany("JavaCorp");
stock.setNumber(250);
ObjectMessage objectMsg = session.createObjectMessage() ;
objectMsg.setObject(stock) ;
queueSender.send(objectMsg) ;
...
```

Deux autres types de messages sont disponibles pour traiter les tableaux de bytes ainsi que les types primitifs, qui sont `BytesMessage` et `StreamMessage`. Ceux-ci utilisent des flux de données, cependant nous ne les étudierons pas dans cet ouvrage.

5.2.3 Modèle de messagerie

JMS offre deux modèles de messagerie : point à point et publication/souscription. De façon simple, le modèle point à point représente une relation « One to One » entre un message et un destinataire alors que le modèle publication/souscription est représenté par une liaison « One to Many ».

Point à point (P2P)

Le modèle point à point permet de connecter les applications clientes entre elles *via* une file d'attente (*queue*). C'est exactement comme le principe d'une pile. Les messages sont envoyés et empilés (fig. 5.2). Lorsqu'une application cliente (consommatrice) est libre, elle reçoit alors les messages empilés.

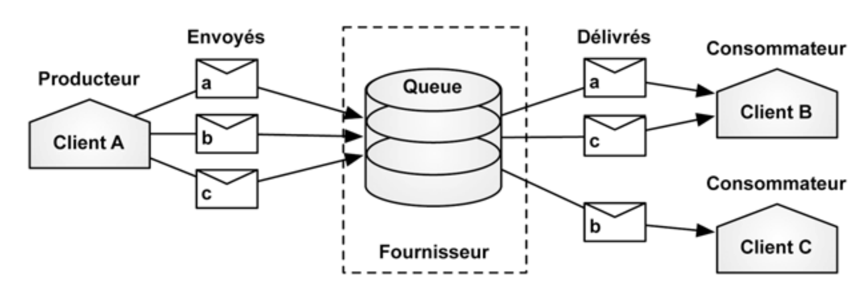


Figure 5.2 — Modèle Point à Point JMS

Dans un modèle point à point un message n'est transmis qu'à une seule application.

Publication/souscription

Dans ce modèle, un producteur peut envoyer un message à plusieurs consommateurs par le biais d'un sujet (*topic*). Chaque consommateur doit cependant s'être préalablement inscrit à ce sujet sinon il ne reçoit rien (fig. 5.3).

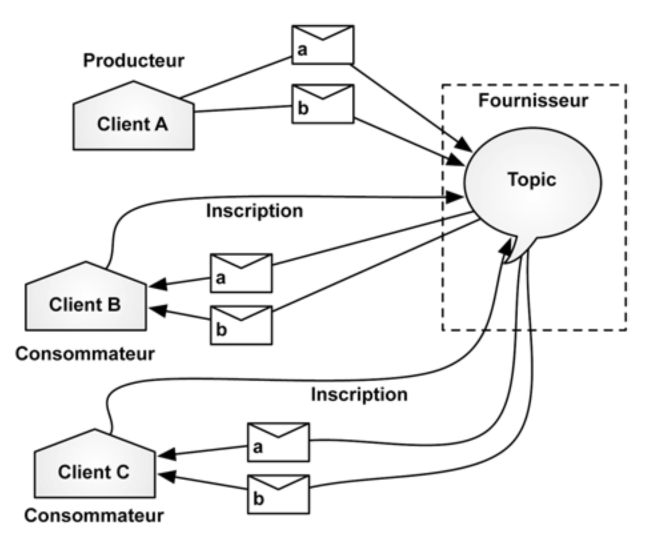


Figure 5.3 — Modèle publication/souscription

Ce modèle peut être assimilé à un *hub* (matériel physique de réseau). Chaque nouveau message est *broadcasté* vers chaque consommateur inscrit. Le producteur est indépendant du fait que le message a bien été reçu par le ou les consommateurs.

Il existe deux types de souscription : temporaire et durable. Dans le premier cas, les consommateurs reçoivent les messages tant qu'ils sont connectés au sujet. Dans le cas d'une souscription durable, on oblige le fournisseur à enregistrer les messages lors d'une déconnexion, et à les envoyer lors de la nouvelle connexion du consommateur.

Quel mode choisir ?

Un problème récurrent pour les architectes est de savoir quel modèle utiliser. En effet, après quelques réflexions, il s'avère qu'il est possible de faire la même chose quel que soit le modèle utilisé. Toutefois, chacun des deux modèles a des particularités intéressantes.

JMS fournit un accès sur ces deux types de modèles car à son développement, certaines entreprises utilisaient le modèle point à point et d'autres utilisaient le modèle d'abonnement.

Dans la plupart des cas, le modèle choisi dépend de l'utilisation que l'on souhaite en faire. On préférera le modèle point à point lorsque l'on souhaite qu'un seul destinataire traite le message et pour être sûr qu'il soit traité. À l'inverse, le modèle d'abonnement est moins restrictif et sert plus particulièrement aux flux d'informations pouvant être utilisés par n'importe quel client.

5.2.4 Applications clientes

Il existe différentes interfaces suivant le modèle de messagerie utilisé. Toutefois une interface commune aux deux modèles existe et fournit un ensemble de méthodes suffisantes dans la majorité des cas !

Client producteur

Voici un exemple d'application cliente qui sert à produire et à envoyer des messages :

```
public class JMSSender {
  public void sendMessage(String text) throws Exception {
    // Initialisation de la connexion
    Context ctx = new InitialContext();
    ConnectionFactory connectionFactory =
      (ConnectionFactory) ctx .lookup("ConnectionFactory");
    Destination destination = (Destination) ctx.lookup("queue/StockValue");
    // Ouverture de la connexion
    Connection cnx = connectionFactory.createConnection();
    // Envoi du message
    Session session = cnx.createSession(false, Session.AUTO_ACKNOWLEDGE);
    MessageProducer producer = session.createProducer(destination);
    TextMessage message = session.createTextMessage();
    producer.send(message);
    // Fermeture de la connexion
    cnx.close();
  }
}
```

Nous retrouvons l'ensemble des composants cités précédemment (`ConnectionFactory`, `Destination`, `Session`...).

Les propriétés pour la création du `Context` JNDI sont dépendantes du fournisseur JMS utilisé. Voici les valeurs utilisées pour GlassFish :

```
Hashtable hashtable = new Hashtable();
hashtable.put("java.naming.factory.initial",
  "com.sun.enterprise.naming.SerialInitContextFactory");
hashtable.put("java.naming.factory.url.pkgs","com.sun.enterprise.naming");
hashtable.put("java.naming.provider.url","iiop://localhost:1050/");
Context jndiContext = new InitialContext(hashtable);
```

Voici celles pour JBoss :

```
Hashtable hashtable = new Hashtable();
hashtable.put("java.naming.factory.initial",
  "org.jnp.interfaces.NamingContextFactory");
hashtable.put("java.naming.factory.url.pkgs","com.sun.enterprise.naming");
hashtable.put("java.naming.provider.url"," localhost:1099");
Context jndiContext = new InitialContext(hashtable);
```

Remarque : ces données sont à titre indicatif et peuvent changer suivant l'évolution des différentes versions de ces serveurs d'application.

Le tableau 5.1 récapitule les interfaces spécifiques qui peuvent être utilisées pour une messagerie de type « point à point ». Le tableau 5.2 récapitule les interfaces spécifiques qui peuvent être utilisées pour une messagerie de type « abonnement ».

Tableau 5.1 — Correspondance entre les interfaces génériques et celles spécifiques aux messageries « point à point »

Générique	Spécifique « point à point »
ConnectionFactory	QueueConnectionFactory
Destination	Queue
Session	QueueSession
MessageProducer	QueueSender

Tableau 5.2 — Correspondance entre les interfaces génériques et celles spécifiques aux messageries par « abonnement »

Générique	Spécifique « abonnement »
ConnectionFactory	TopicConnectionFactory
Connection	TopicConnection
Destination	Topic
Session	TopicSession
MessageProducer	TopicPublisher

Client consommateur

Le code du consommateur est très similaire au code précédent. En effet, les seules choses qui diffèrent sont la récupération d'un `MessageConsumer` et l'envoi du message.

```
public class JMSReceiver {

  public void receiveMessage() throws Exception {
    // Initialisation de la connexion
    Context ctx = new InitialContext();
    ConnectionFactory connectionFactory = (ConnectionFactory) ctx
        .lookup("ConnectionFactory");
    Destination destination = (Destination) ctx
        .lookup("queue/StockValue");
```

```
    Connection cnx = connectionFactory.createConnection();
    Session session = cnx.createSession(false,  Session.AUTO_ACKNOWLEDGE);
    // Création d'un consommateur de messages
    MessageConsumer consumer = session.createConsumer(destination);
    consumer.setMessageListener(new MessageListener() {
      public void onMessage(Message message) {
        System.out.println(message);
      }
    });
    // Ouverture de la connexion
    cnx.start();
  }
}
```

Dans cet exemple, nous utilisons le concept de *listener* évitant tout blocage de l'application lors de la réception d'un message. Pour cela, nous devons implémenter l'interface `MessageListener`. La méthode `onMessage(Message message)` est alors appelée automatiquement lors de la réception d'un message.

Attention : n'oubliez pas de démarrer la connexion avec la méthode `start()` sinon aucun message ne sera reçu.

De la même façon qu'avec le producteur, le consommateur peut utiliser des interfaces plus spécifiques pour chaque modèle de messagerie. Le tableau 5.3 récapitule les interfaces spécifiques qui peuvent être utilisées pour une messagerie de type « point à point ». Le tableau 5.4 récapitule les interfaces spécifiques qui peuvent être utilisées pour une messagerie type « abonnement ».

Tableau 5.3 — Correspondance entre les interfaces génériques et celles spécifiques aux messageries « point à point »

Générique	Spécifique « point à point »
MessageConsumer	QueueReceiver

Tableau 5.4 — Correspondance entre les interfaces génériques et celles spécifiques aux messageries par « abonnement »

Générique	Spécifique « abonnement »
MessageConsumer	TopicSuscriber

5.2.5 Session Bean et JMS

Nous avons vu comment créer une application cliente « producteur » et une application cliente « consommateur ». Toutefois, ces applications s'exécutent indépendamment de tout système d'entreprise et sont très limitées.

Quelle est la meilleure solution pour exécuter une méthode métier lors de la réception d'un message ? Une première solution serait de se baser sur les exemples précédents et d'appeler une méthode d'un Session Bean lors de la réception d'un message (fig. 5.4).

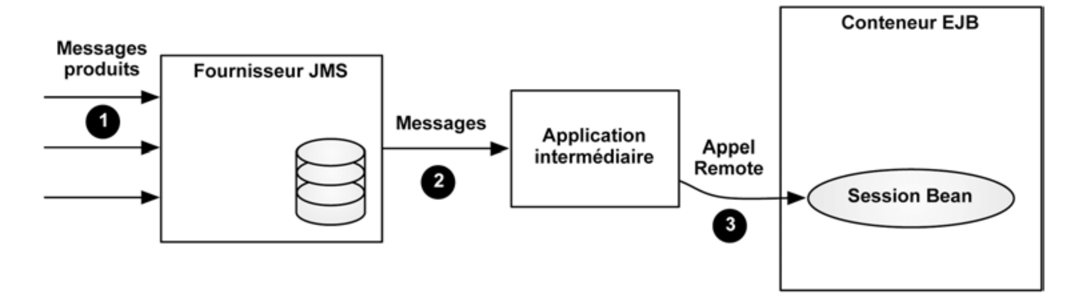


Figure 5.4 — Première solution

Cette solution, digne d'un bricoleur, n'est guère fiable. En effet, si l'application intermédiaire entre le fournisseur JMS et le conteneur EJB s'arrête, toute la chaîne s'effondre.

Une autre solution consisterait à tenter d'intégrer l'écoute JMS directement au sein d'un Session Bean (fig. 5.5). Cela évite l'utilisation d'un *proxy* instable entre le fournisseur JMS et le conteneur EJB. Cependant un problème majeur existe : Comment un Session Bean peut-il attendre les messages ? L'exécution d'une de ses méthodes ne peut être lancée par une application cliente… Imaginons qu'un client appelle une méthode du Session Bean qui écoute sur une file d'attente.

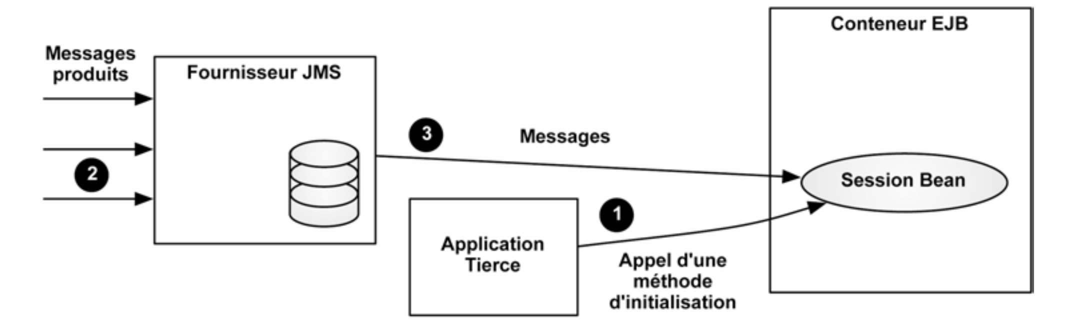


Figure 5.5 — Seconde solution

Dans ce cas, la méthode `receive()` de l'objet `MessageConsumer` bloque le thread d'exécution du Session Bean tant qu'un message n'est pas arrivé. Si aucun message n'arrive, le thread se bloque indéfiniment.

La solution préconisée, et la plus facile à mettre en place, est l'utilisation d'un MDB pour l'exécution de code métier lors de la réception d'un message (fig. 5.6).

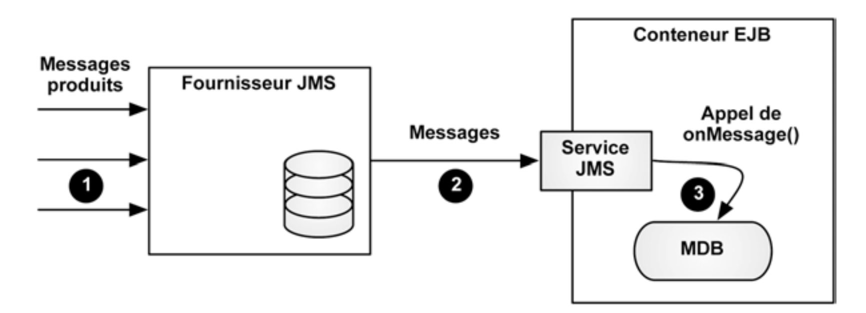


Figure 5.6 — Solution préconisée

Dans le cas d'un MDB, la connexion au fournisseur JMS est établie par le conteneur. Le développeur est sûr de travailler dans un environnement stable et n'a pas besoin de se préoccuper des soucis techniques liés à la connexion.

5.3 MESSAGE DRIVEN BEAN

5.3.1 Rôle d'un MDB

Un MDB est un composant permettant de traiter les messages arrivant sur une destination. Il est généralement considéré comme *listener* JMS. Un MDB offre un accès à sa logique métier par l'intermédiaire de messages. La figure 5.7 présente un cas d'utilisation pour ce genre de pratique.

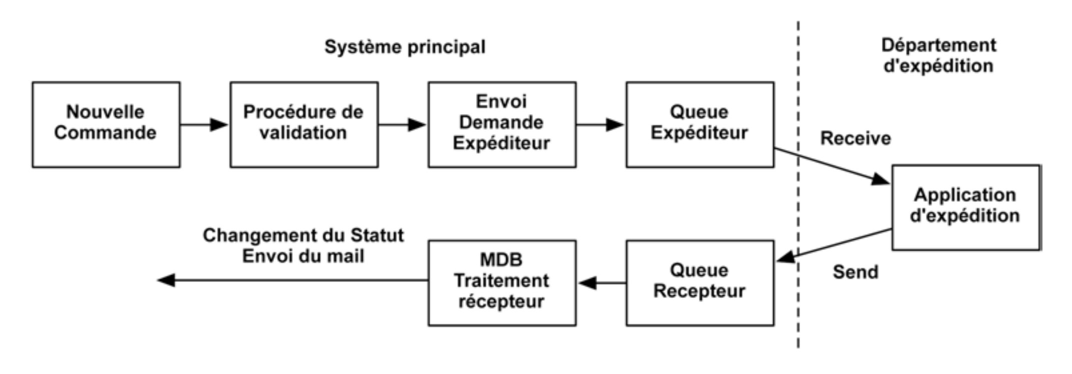


Figure 5.7 — Cas d'utilisation JMS et MDB

Pour comprendre cet exemple, il faut se mettre dans le contexte d'une entreprise de vente par correspondance par exemple. Lorsqu'une commande est créée, elle passe par un ensemble de validations (bancaire, postale ...). Une fois la commande examinée, un message est alors envoyé au département « expédition ». Ce département est constitué de plusieurs personnes préparant les expéditions. Il est donc intéressant d'utiliser un modèle « point à point » afin d'être sûr que les messages sont traités par une entité unique. De plus, la répartition des messages est gérée automatiquement par JMS.

L'idée est simple, chaque personne a une instance d'un programme présentant la commande à expédier. Ce programme est connecté à la queue JMS et récupère un message (une expédition donc). Une fois le colis préparé et envoyé, l'application renvoie un message vers une queue de réponse JMS afin d'indiquer l'envoi. Ce message est alors récupéré *via* un MDB et traité (changement du statut de la commande).

Un MDB peut également être utilisé pour mettre à jour des données. Dans notre application d'exemple, « StockManager », nous utilisons ce composant pour écouter les messages contenant les nouvelles valeurs boursières (venant d'une destination de type *topic*). De cette façon, il met à jour la base de données avec les nouvelles valeurs des actions (fig. 5.8).

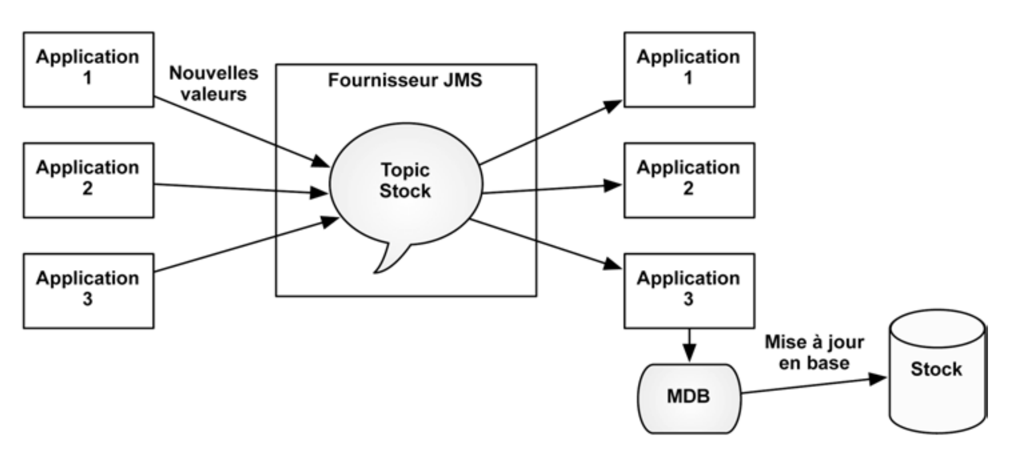


Figure 5.8 — Cas d'utilisation d'un MDB de mise à jour.

5.3.2 EJB 2 : écriture d'un MDB

La classe du MDB

Tout *Message Driven Bean* doit implémenter l'interface `javax.ejb.MessageDrivenBean`. Cette interface définit les méthodes liées à son intégration dans le conteneur EJB. Elle contient deux méthodes.

```
public interface MessageDrivenBean extends EnterpriseBean {
  public void setMessageDrivenContext(MessageDrivenContext ctx)
    throws EJBException;
```

```
  public void ejbRemove()
               throws EJBException;
}
```

- `setMessageDrivenContext (MessageDrivenContext)` : cette méthode est appelée après l'instanciation du MDB. Elle lui permet de conserver une référence au contexte lié au conteneur pour ensuite agir avec le conteneur EJB.
  Nous étudions plus en détail cet objet dans la partie EJB 3. En effet, cet objet n'a pas eu de modification entre les deux versions et son fonctionnement est resté le même.
- `ejbRemove()` : cette méthode est appelée avant l'arrêt du MDB. Ceci se produit quand le conteneur juge nécessaire de retirer l'instance de son pool (lors de la baisse du nombre de requêtes ou de la fermeture de l'application).
  Dans le cadre d'utilisation de JMS, il est obligatoire d'implémenter l'interface `javax.jms.MessageListener`. Elle peut être assimilée à l'interface métier d'un Session Bean. Elle définit une unique méthode : `onMessage(Message message)` qui est automatiquement appelée par le conteneur lors de la réception d'un message.

```
public class StockValueListenerBean implements MessageDrivenBean,

  MessageListener {
  private MessageDrivenContext ctx;
  public void onMessage(Message message) {
    // traite le message ...
  }
  public void setMessageDrivenContext(MessageDrivenContext ctx)
    throws EJBException {
    this.ctx = ctx;
  }
  public void ejbRemove() throws EJBException {
    // libère les ressources éventuelles
  }
}
```

Dans l'exemple précédent, la classe du Session Bean implémente les interfaces `MessageDrivenBean` et `MessageListener`. Le traitement du message est effectué dans la méthode `onMessage()`. Dans notre exemple, le message contient la nouvelle valeur d'une action (représentée par son code unique).

Le descripteur de déploiement

Comme tout autre type d'EJB, il est nécessaire de renseigner le descripteur de déploiement « ejb-jar.xml » afin d'informer le conteneur de la disponibilité de l'EJB et de la manière de le déployer. Dans le cas d'un MDB, celui-ci doit être associé à une `Destination` afin de recevoir des messages.

```
<message-driven>
  <description><![CDATA[MDB de mise à jour des valeurs boursières]]></
description>

  <ejb-name>StockValueListener</ejb-name>
```

```
  <ejb-class>
    com.labosun.stockmanager.ejb2.mdb.StockValueListenerBean
  </ejb-class>

  <messaging-type>javax.jms.MessageListener</messaging-type>
  <transaction-type>Container</transaction-type>
  <message-destination-type>javax.jms.Queue</message-destination-type>
  <activation-config>
    <activation-config-property>
     <activation-config-property-name>
        destinationType
     </activation-config-property-name>
     <activation-config-property-value>
        javax.jms.Queue
     </activation-config-property-value>
    </activation-config-property>
    <activation-config-property>
      <activation-config-property-name>
        acknowledgeMode</activation-config-property-name>
      <activation-config-property-value>
        Auto-acknowledge
      </activation-config-property-value>
    </activation-config-property>
  </activation-config>
</message-driven>
```

La balise `<messaging-type>` définit quel est le fournisseur de message (ici JMS car le MDB implémente l'interface `javax.jms.MessageListener`). La balise `<message-destination-type>` permet, quant à elle, de spécifier le type de destination à utiliser pour JMS (ici une file d'attente : `Queue`).

La déclaration des paramètres de la destination utilisée s'opère *via* les propriétés de configuration `<activation-config-property>` incluses dans les balises `<activation-config>`.

Nous sommes ici restés très succincts concernant les possibilités des MDB. Il est également possible de définir le renvoi d'accusés de réception, ainsi que des critères de sélection sur les messages… Ces éléments vont, cependant, être vus dans la partie suivante.

5.3.3 EJB 3 : écriture d'un MDB

La classe du Bean

En EJB 3, il n'est maintenant plus obligatoire d'implémenter l'interface `MessageDrivenBean`. Seule l'annotation `@MessageDriven` est nécessaire pour définir votre classe en tant que MDB. L'implémentation de `MessageListener` est, quant à elle, nécessaire si le MDB travaille avec JMS.

```
@MessageDriven( name = "StockValueListener"
activationConfig = {
  @ActivationConfigProperty(propertyName = "destinationType",
```

```
      propertyValue = "javax.jms.Topic"),
    @ActivationConfigProperty(propertyName = "destination",
      propertyValue = "topic/StockValueTopic")}
)
public class StockValueListenerBean implements MessageListener {

  @Resource
  private MessageDrivenContext ctx;

  public void onMessage(Message message) {
    // update stock value procedure
  }
}
```

Il n'est désormais plus obligatoire d'utiliser le descripteur de déploiement pour configurer votre MDB. Tout est faisable *via* les annotations.

L'annotation `@MessageDriven` permet de définir le nom du MDB au sein du conteneur (attribut `name`). L'attribut `activationConfig` permet de configurer les propriétés du MDB. Dans l'exemple précédent, nous définissons le type de destination `destinationType` et le nom JNDI de la destination utilisée (`destination`). Pour cela, nous utilisons un tableau de `@ActivationConfigProperty` précisant le nom de la propriété `propertyName` et sa valeur `propertyValue`.

> **Attention :** l'attribut `mappedName` n'est pas à utiliser car il n'est pas portable entre les différents serveurs d'applications.

Il est possible de récupérer le contexte du MDB *via* l'injection (annotation `@Resource`). Le type de cet objet est `MessageDrivenContext`. Nous détaillerons son utilisation plus loin.

Vous pouvez remarquer que le développement de MDB est vraiment simplifié. Nous allons maintenant étudier les différents points de détails de ces composants.

MessageDrivenContext

Comme pour les Session Beans, les Message Driven Beans ont également un contexte d'exécution de type `MessageDrivenContext`. Il est similaire à l'objet `SessionContext` expliqué dans la partie Session Bean. En effet, cette interface hérite simplement de `EJBContext` et ne rajoute aucune méthode.

L'utilisation de cet objet est cependant particulière pour les MDB. Les méthodes `getEJBHome()` et `getEJBLocalHome()` lancent une `RuntimeException` si elles sont invoquées. En effet, un MDB ne possède pas d'interface *home*. De même pour les méthodes `getCallerPrincipal()` et `isCallerInRole()` car un MDB n'est pas appelé par une personne (*a caller*) et donc n'a aucun contexte de sécurité associé.

Pour des détails concernant la gestion des transactions dans un MDB, veuillez vous référer au chapitre 9.

Tolérance de la déconnexion à un « topic »

Le modèle de messagerie de type « abonnement » oblige les applications clientes à être connectées au sujet (*topic*) pour recevoir les messages de celui-ci. Si un problème de connexion survient, les clients déconnectés perdent les messages émis durant leur déconnexion.

Cependant, comme nous l'avions précisé au début de ce chapitre, le mode *topic* offre la possibilité d'utiliser un abonnement durable. L'intérêt est donc de pouvoir recevoir les messages émis depuis la dernière déconnexion.

L'utilisation de ce genre d'abonnement doit être précisée au niveau des propriétés du MDB. La propriété à utiliser est `subscriptionDurability`. Les valeurs prises par celle-ci sont : `Durable` ou `NonDurable`. Par défaut, une souscription est `NonDurable`.

```
@MessageDriven(
  activationConfig={
...
  @ActivationConfigProperty(
     propertyName="subscriptionDurability", propertyValue="Durable")
}
)
...
```

D'autres propriétés existent et peuvent être utilisées (`subscriptionName`, `clientId`…).

Lorsque la destination est de type `Queue`, le principe d'abonnement durable n'a aucun sens. Par nature pour ce genre de destination, le facteur « durable » n'a pas d'importance car les messages sont automatiquement stockés et doivent être consommés par un client unique.

Sélecteur de message

Il est possible de préciser certains critères permettant de ne pas recevoir l'ensemble des messages d'une destination. Le sélecteur de message utilise les propriétés du message en tant que critère dans les expressions conditionnelles. Ces conditions utilisent des expressions booléennes afin de déterminer les messages à recevoir. Voici un exemple d'utilisation de ces sélecteurs de message.

```
@ActivationConfigProperty ( propertyName = "messageSelector",
  propertyValue = "senderType = 'StockSenderCorp'" )
```

Ces sélecteurs se basent sur les propriétés des messages. Celles-ci se situent dans l'en-tête du message, donc dépendant du contenu, et sont assignées par le créateur du message. Tous les types de message intègrent les méthodes de lecture et d'écriture de propriétés. En effet, ces méthodes sont décrites dans l'interface `javax.jms.Message` (super-interface définissant un message). Les types de propriétés se basent sur les primitives Java : `boolean`, `int`, `short`, `char`…

Pour définir les valeurs des propriétés il faut utiliser les méthodes `setXxxProperty(Xxx)` où `Xxx` représente les types `byte`, `float`, `String`, `Object`... Les méthodes `getXxxProperty()` sont également proposées pour récupérer les valeurs. Dans l'exem-

ple suivant, nous définissons des propriétés sur lesquelles nous pourrons définir des critères de récupération.

```
//...
TextMessage message = session.createTextMessage();
message.setText("Les indices boursiers à Paris ont gagné 4 % aujourd'hui") ;
message.setIntProperty("result",4);
message.setStringProperty("place","FR/Paris");
message.setStringProperty("senderType","StockSenderCorp");
//...
```

Grâce aux propriétés que nous venons de définir, nous pouvons appliquer des critères de sélection.

```
@MessageDriven(activationConfig={
  activationConfig= {
  @ActivationConfigProperty(
   propertyName = "messageSelector",
   propertyValue = "place = 'FR/Paris' and result > 0")}
}
```

Le système repose sur les mêmes concepts que la sélection des enregistrements avec SQL. Vous pouvez utiliser les opérateurs `AND`, `OR`, `<`, `>`… D'autres fonctionnalités sont possibles. Prenons le cas où nous souhaitons traiter dans un MDB tous les messages ayant un rapport avec les places financières parisienne, londonienne, et francfortoise.

```
place IN ('FR/Paris', 'UK/London', 'GER/Francfort')
```

On peut également penser à un système d'alerte, dans le cas où le résultat d'une journée n'est pas situé entre +10 % et -10 %, pour ainsi prévenir les investisseurs qu'il s'agit d'une journée sortant de l'ordinaire.

```
result NOT BETWEEN -10 AND 10
```

Ou encore le traitement des places boursières française ou anglaise.

```
place LIKE 'FR/%' OR place LIKE 'UK/%'
```

Cette fonctionnalité est utile lorsque l'on souhaite trier et répartir les messages situés dans la même destination vers différents MDB, afin de leur appliquer des traitements différents.

Accusé de réception

L'inconvénient du traitement asynchrone des messages est qu'il n'y a pas de valeur de retour à l'expéditeur. Il est donc difficile pour lui de savoir si le message a bien été transmis (lorsqu'il souhaite le savoir, bien entendu). Il existe différentes solutions à ce problème.

La première consiste à utiliser des accusés de réception, mécanisme transparent géré par le fournisseur et le conteneur MDB. Cela permet à l'application cliente de signaler au fournisseur que le message a bien été reçu. Sans cet accusé de réception, le message continuera d'être envoyé.

Ce mécanisme repose sur les transactions au niveau du MDB. Lorsque celle-ci est gérée par le conteneur, l'accusé est envoyé à la suite du commit ou non si la transaction échoue.

Il existe deux modes d'accusé de réception : « Auto-acknowledge » et « Dups-ok-acknowledge ». Le mode « Auto-acknowledge » définit que l'accusé de réception doit être envoyé dès que le message a été transféré au MDB.

```
@MessageDriven(
  name="MyQueue",
  mappedName = "queue/MyQueue",
  activationConfig={
    @ActivationConfigProperty(
      propertyName="acknowledgeMode",propertyValue="Auto-acknowledge")
  }
)
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class MyTopicListener implements MessageListener {
//...
}
```

Dans le cas d'Auto-acknowledge, cela définit un envoi automatique lors du « *commit* » de la transaction. Voici le code utilisé pour le second mode :

```
@MessageDriven(
    name="MyQueue",
    mappedName = "queue/MyQueue",
    activationConfig={
       @ActivationConfigProperty(
      propertyName="acknowledgeMode",propertyValue="Dups-ok-acknowledge")
     }
)
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class MyTopicListener implements MessageListener {
//...
}
```

Pour Dups-ok-acknowledge, l'accusé de réception est repoussé à un instant indéterminé et le conteneur choisit ensuite le moment où il a peu de tâches à traiter pour l'envoyer. Cela permet évidemment d'économiser les ressources. On peut cependant déconseiller cette dernière valeur car le fournisseur pourrait croire que le message n'a pas été traité et déciderait de le transmettre à nouveau (ce qui pourrait entraîner des disfonctionnements). De plus, le coût d'envoi d'un accusé de réception est négligeable que ce soit en capacité de calcul ou en charge réseau.

La deuxième solution consiste à spécifier la valeur JMSReplyTo dans les paramètres d'en-tête du message. Cela permet au destinataire d'envoyer une réponse vers la destination paramétrée. Du côté de l'expéditeur, on définit la destination de réponse de la manière suivante :

```
//...
Destination replyDestination = context.lookup("queue/ReplyQueue");
message.setJMSReplyTo(replyDestination);
```

Le MDB recevant le message peut ensuite récupérer cette propriété et envoyer à son tour un message.

```
Destination replyDestination = message.getReplyTo();
//...
```

Cette méthode diffère de la précédente car elle permet un réel retour d'information concernant le traitement du message. Cette solution est typiquement utilisée dans notre exemple impliquant des expéditions. Le message de retour alerte le système lorsque l'objet est préparé et expédié. Cela n'aurait pas pu être implémenté avec la première solution. Cette solution est également intéressante pour remonter des rapports d'erreurs dans le processus métier.

MDB et client JMS

Nous avons orienté notre présentation MDB et JMS majoritairement sur la consommation de messages plus que sur la production de ces messages. Il existe cependant de nombreuses évolutions facilitant le développement de l'envoi de ces messages.

Rappelons les différentes étapes devant être mises en place pour l'envoi d'un message JMS :

- Récupération d'un objet `ConnectionFactory`,
- Récupération de la destination,
- Ouverture d'une connexion,
- Création d'une session,
- Création d'un `MessageProducer`,
- Création d'un `Message`,
- Envoi du message,
- Déconnexion.

```
public class StockAlertListener {
  @Resource (mappedName = "ConnectionFactoryName")
  private ConnectionFactory cnxFactory;

  @Resource (mappedName = "queue/StockAlertQueue")
  private Queue stockAlertQueue;

  public void sendAlertMessage(String textAlert) {
    Connection cnx = cnxFactory.createConnection();
    Session session = cnx.createSession(true, 0);
    MessageProducer producer = session.createProducer(stockAlertQueue);
    TextMessage message = session.createTextMessage();
    message.setText(textAlert);
    producer.send(message);
    cnx.close();
  }
//...
```

Les deux premières étapes de récupération sont automatisées grâce au principe d'injection. Pour cela, nous avons utilisé l'annotation `@Resource` qui définit le nom JNDI de la `ConnectionFactory` et de la `Destination` (de type `Queue`, ici).

Le code de la méthode `sendAlertMessage(String textAlert)` ne change guère par rapport au code de l'application cliente présenté auparavant.

> **Remarque :** les exemples présentés dans cette partie peuvent tout à fait être implémentés au sein d'un Session Bean exactement de la même manière.

Cycle de vie

Tout comme les Session Beans, un Message Driven Bean possède un cycle de vie géré par le conteneur. Ce type de composant ne possède que deux états : « N'existe pas » et « Méthode prête ».

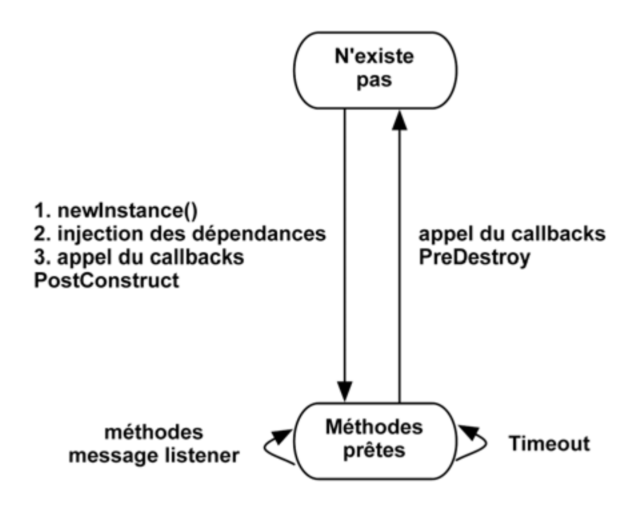


Figure 5.9 — Cycle de vie d'un MDB

Un MDB est dans l'état « N'existe pas » lorsqu'il n'existe aucune instance en mémoire de celui-ci (il n'a pas encore été instancié). Un MDB est dans l'état « Méthode prête » lorsque l'instance est prête à l'emploi. Une fois l'application démarrée, le conteneur crée un ensemble d'instances de MDB dont l'état est « Méthode prête ».

Les étapes suivantes décrivent les changements d'état que peut subir un MDB :

- L'instance est créée lorsque le conteneur appelle la méthode `newInstance()` dans la classe du composant (généralement au démarrage de l'application). Ensuite, le conteneur injecte toutes les dépendances de l'objet. Ces dépendances rassemblent les injections annotées avec `@Resource`, `@EJB` … Le conte-

neur appelle ensuite les méthodes annotées avec `@PostConstruct`. L'instance est maintenant prête à traiter les messages délivrés à la destination écoutée : état « Méthode prête ».

- Lorsque le conteneur n'a plus besoin de l'instance, il appelle les méthodes annotées avec `@PreDestroy`. Cela se produit généralement lorsque la réserve d'instances devient disproportionnée par rapport aux besoins ou que l'application s'arrête.

L'intérêt de la méthode déclarée avec `@PostConstruct` est d'ouvrir une connexion vers un service, comme l'accès à un annuaire d'entreprise ou plus simplement à un fichier. La seconde méthode, annotée de `@PreDestroy`, est utilisée pour libérer les ressources utilisées.

```
...
private FileWriter fw = null;

@PostConstruct
private void connectToFile() {
  try {
   fw = new FileWriter(new File("save_topic.txt")) ;
  }
  catch(Exception e) { ... }
}

@PreDestroy
private void unconnectToFile() {
  try {
   fw.close();
  }
  catch(Exception e) { ... }
}

public void onMessage(Message message){
  try{
//...
   fw.flush();
  }catch(IOException ex){
  ex.printStackTrace();
  }
}
```

Remarquez que nous avons forcé l'écriture dans le fichier à la fin de la méthode `onMessage()` avec `FileWriter.flush()`. L'intérêt est d'être sûr que le fichier soit écrit. En effet, pour certaines raisons, les méthodes de rappel peuvent ne pas être appelées et les données restées en mémoire peuvent être perdues.

En résumé

Les *Message Driven Beans* gagnent en simplicité de mise en place et en maniabilité, principalement grâce aux annotations.

L'interconnexion entre les systèmes étant, aujourd'hui, de plus en plus exploitée, les MDB s'intègrent parfaitement dans de telles architectures (SOA).

6

L'unité de persistance

Objectif

Partie intégrante du *framework* Java EE 5, l'unité de persistance est la « boîe noire » qui permet de rendre persistants les Entity Beans. Plus qu'un simple fournisseur de persistance, celle-ci va permettre aux développeurs d'optimiser leurs applications selon la gestion de leurs Entity Beans.

L'unité de persistance est l'élément clé de la gestion des Entity Beans au sein d'une application.

Après avoir présenté l'unité de persistance, nous étudierons comment l'utiliser au sein d'applications entreprises.

6.1 LE SYSTÈME DE PERSISTANCE : UNE ÉVOLUTION MAJEURE

Suite à l'attrait grandissant pour certains concepts mis en place par la communauté, Sun a décidé de garder, pour la spécification EJB 3, les meilleurs aspects de la spécification EJB 2 et d'y intégrer de nouveaux concepts. L'ensemble des aspects de persistance ont été réunis dans leur propre spécification : « Persistence API 1.0 ». Cette API fournit une couche d'abstraction de plus haut niveau que celle de JDBC afin d'être totalement indépendante des spécificités des fournisseurs de persistance.

Les changements entre les versions 2 et 3 des EJB sont considérables. Le système de gestion de persistance ayant été totalement revu, les seules similitudes se retrouvent sur seulement deux points :

- Un langage pour effectuer des requêtes : l'EJB-QL.
- Un concept de requête prédéfinie lors de la compilation (bien que de grandes différences existent cependant).

La principale évolution réside, cependant, dans le fait que les Entity Beans sont maintenant des objets simples (POJO). Ils sont désormais *managés* par une unité de persistance qui permet d'intégrer l'utilisation de ces Entity Beans au sein d'applications Java EE et même Java SE.

6.2 QU'EST-CE QU'UNE UNITÉ DE PERSISTANCE ?

Bien que la mise en place d'Entity Beans EJB 3 au sein d'une application soit assez simple, la persistance des informations doit être configurée pour permettre de sauvegarder les données dans la ou les source(s) de données.

La gestion de la persistance avec EJB 2, bien que directement liée aux classes d'Entity Beans, vous oblige à suivre un ensemble de règles complexes dictées par la spécification (voir paragraphe 4.2). Avec la spécification EJB 3, les interfaces *home* n'existant plus, comment pouvons-nous alors travailler sur nos Entity Beans ? Comment sont-ils créés et mis à jour ?

Pour éviter ces procédures complexes, la communauté Java s'est alors tournée vers des solutions tierces plus « légères », comme Hibernate (*www.hibernate.org*), TopLink (*www.oracle.com*), iBatis (*ibatis.apache.org*) ou bien d'autres. Communément appelés *frameworks*, ils permettent de gérer la persistance des données en effectuant un « mapping objet/relationnel » de façon simple.

C'est devant l'attrait de la communauté pour ces *frameworks*, que la spécification EJB 3 a évolué dans cette direction, en intégrant l'API « Persistence 1.0 ». La solution adoptée pour la gestion de ces Entity Beans est l'utilisation d'une unité de persistance (ou contexte de persistance) qui prend en charge la sauvegarde des informations dans la source de données, de manière autonome.

Une unité de persistance (*Persistence Unit*) est caractérisée par les points suivants :

- un ensemble d'Entity Beans,
- un fournisseur de persistance (*Provider*),
- une source de données (*Datasource*).

Une entité persistante étant vouée à être enregistrée dans une source de données, le rôle de l'unité de persistance est :

- De savoir où et comment stocker les informations.
- De s'assurer de l'unicité des instances de chaque identité persistante.

- De gérer les instances et leur cycle de vie : c'est le gestionnaire d'entité (*Entity Manager*).

Le cycle de vie et les différents états des Entity Beans sont expliqués à la fin de ce chapitre. Toutefois, il est important de comprendre la différence entre l'état attaché (*attached*) et détaché (*detached*) avant de rentrer dans les détails.

Quand un Entity Bean est attaché à un contexte de persistance, les modifications appliquées à l'objet sont alors automatiquement synchronisées avec la base de données, *via* l'Entity Manager. À l'inverse, un Entity Bean est dit détaché lorsqu'il n'a plus aucun lien avec l'Entity Manager.

Remarque : nous expliquerons le passage d'un état à l'autre au fur et à mesure des exemples. La dernière partie résume le cycle de vie d'un Entity Bean.

En résumé, l'unité de persistance est un groupe d'instance d'entités *managées* dont leur gestion est réalisée par l'Entity Manager.

6.3 INTÉGRATION ET PACKAGING D'UNE UNITÉ DE PERSISTANCE

Grâce à cette nouvelle spécification et à ses conteneurs légers, l'utilisation des Entity Beans n'est plus fermée au monde J2EE ou à un conteneur EJB. Vous pouvez désormais « déployer » des Entity Beans dans de nombreuses applications :

- Application Entreprise (EAR).
- Module Java Bean Entreprise (EJB-JAR).
- Application web (WAR).
- Client d'application entreprise (JAR).
- Un environnement Java SE compatible.

Ce dernier point est, sans doute, le plus appréciable, en effet il vous permet d'inclure facilement et simplement un système de persistance de données à une application Java SE.

Dans les parties suivantes nous allons voir la configuration et l'utilisation d'une unité de persistance au sein d'un module EJB 3 d'une application. Notez qu'au sein d'une telle application, cette unité de persistance est exploitée *via* un Session Bean.

6.3.1 Paramétrage de l'unité de persistance

La nouvelle spécification définit un fichier qui regroupe l'ensemble des informations de persistance. Vous devez nommer ce fichier : « persistence.xml » et placer ce fichier dans le répertoire « META-INF », à la racine du projet.

Ce fichier sera lu par le conteneur EJB lors du déploiement de l'application. Le conteneur créé alors une instance du fournisseur de persistance avec les paramètres demandés.

Attention : vous devez nommer correctement le fichier « persistence.xml ». Si le nom ne correspond pas, le conteneur n'associera aucun contexte de persistance à l'application.

Voici le schéma descriptif (DTD, *Document Type Definition*) à laquelle le fichier persistence.xml obéit :

```
<!ELEMENT persistence (persistence-unit*)>
  <!ELEMENT persistence-unit (description?,provider?,jta-datasource?,
      non-jta-datasource?,(class|jar-file|mapping-file)*,
      execlude-unlisted-classes?,properties?)>
    <!ELEMENT description (#PCDATA)>
    <!ELEMENT provider (#PCDATA)>
    <!ELEMENT jta-datasource (#PCDATA)>
    <!ELEMENT non-jta-datasource (#PCDATA)>
    <!ELEMENT mapping-file (#PCDATA)>
    <!ELEMENT jar-file (#PCDATA)>
    <!ELEMENT class (#PCDATA)>
    <!ELEMENT exclude-unlisted-classes EMPTY>
    <!ELEMENT properties (property*)>
    <!ELEMENT property EMPTY>

    <!ATTLIST persistence-unit name CDATA #REQUIRED>
    <!ATTLIST property name CDATA #REQUIRED>
    <!ATTLIST property value CDATA #REQUIRED>
```

La balise racine `<persistence>` ne contient que des balises `<persistence-unit>`. Voici une description de chacune des balises utilisées, ainsi que le nombre de paramètres (entre parenthèses) qu'elles peuvent prendre :

- `<persistence-unit>` (0-N) : déclare une unité de persistance.
  - L'attribut `name` affecte un nom unique à cette unité dans votre application. Le nom est utilisé pour identifier l'unité de persistance lors de son utilisation avec les annotations `@PersistenceContext` et `@PersistenceUnit` pour la création d'`EntityManager` ou d'`EntityManagerFactory` (voir paragraphe 6.5).
  - L'attribut `type` définit si l'unité de persistance est gérée et intégrée dans une transaction Java EE (JTA) ou si vous souhaitez gérer de façon manuelle les transactions (RESOURCE_LOCAL) *via* l'Entity Manager. La valeur par défaut en environnement Java EE est JTA et RESOURCE_LOCAL en environnement Java SE. Des détails concernant ce type sont donnés plus loin.

Vous devrez définir plusieurs unités de persistance si vous souhaitez utiliser plusieurs sources de données.

Nous allons maintenant décrire les balises qui permettent de paramétrer une unité de persistance. Vous devez donc placer ces balises à l'intérieur d'une balise `persitence-unit` :

- `<description>` (1) : permet de décrire l'unité de persistance courante.
- `<provider>` (0-1) : permet de définir la classe d'implémentation du fournisseur de persistance utilisé dans l'application (sous-classe de `javax.persistence.spi.PersistenceProvider`). Chaque unité de persistance utilise un fournisseur unique. Vous pouvez cependant, en environnement Java EE, utiliser le fournisseur par défaut du serveur d'applications :
  - JBoss utilise Hibernate : `org.hibernate.ejb.HibernatePersistence`
  - Oracle Application Server utilise TopLink : `oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider`
  - GlassFish (Sun Application Server 9) utilise TopLink : `oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider`
  - KODO : `kodo.persistence.PersistenceProviderImpl`
  - En environnement Java SE, vous devez spécifier le fournisseur de persistance.
- `<jta-datasource>` (0-1) : permet de définir le nom JNDI de la source de données transactionnelle à utiliser. Celle-ci doit être paramétrée sur le serveur et prendre en compte les transactions.
- `<non-jta-datasource>` (0-1) : permet de définir le nom JNDI de la source de données non transactionnelle à utiliser. Celle-ci doit être paramétrée sur le serveur. En environnement Java SE, vous ne pouvez pas utiliser de source de données. Il faut utiliser les propriétés (balise `<properties>`) du provider pour spécifier la base de données à utiliser.
- `<mapping-file>` (0-N) : permet de définir le fichier de *mapping* XML pour les Entity Beans (lorsqu'il n'est pas possible d'utiliser les annotations).
- `<properties>` (0-N) : Cette balise vous permet de configurer les attributs de configuration du fournisseur de persistance. Cette balise regroupe l'ensemble des propriétés `<property>`. Une `<property>` est définie par un nom (attribut `name`) et une valeur (attribut `value`).

Conseil : les serveurs d'applications fournissant, la plupart du temps, des paramétrages par défaut pour le moteur de persistance, il peut être important de consulter ceux-ci ou de les éditer. Ces paramétrages sont situés dans le fichier : « /server/defaultdeploy/ejb3.deployer/META-INF/persistence.properties », du répertoire JBoss.

Une unité de persistance est *mappée* à un ensemble d'Entity Bean. Par défaut dans un environnement Java EE, le conteneur analyse l'ensemble des classes du fichier JAR contenant le fichier « persistence.xml ». Pour définir manuellement les classes à *mapper*, vous pouvez utiliser les balises suivantes :

- `<jar-file>` (0-N) : permet de définir le(s) fichier(s) JAR, contenant les classes des Entity Beans, à analyser.

- `<class>` (0-N) : permet de définir les noms des classes à intégrer dans l'unité. En environnement Java SE, vous devez définir l'ensemble des classes d'entités utilisées.
- `<exclude-unlisted-classes>` (0-1) : indique que les classes non listées explicitement ne doivent pas être intégrées à cette unité. En environnement Java EE, le conteneur cherche automatiquement les classes annotées `@Entity` et les ajoute aux unités de persistance. Vous devez utiliser la balise `exclude-unlisted-classes` lorsque plusieurs unités de persistance sont définies et qu'elles utilisent des ensembles distincts d'Entity Bean dans une même application.

Finalement, l'ensemble des classes à *mapper* est défini par l'ensemble des :

- Classes annotées avec `@Entity` incluses dans les fichiers JAR définis *via* `<jar-file>`.
- Classes mappées dans le fichier « META-INF/orm.xml » s'il existe ou dans n'importe quel fichier XML défini *via* `<mapping-file>`.
- Classes listées *via* `<class>`.

Généralement, vous n'avez pas à utiliser les balises `<jar-file>` et `<class>` sauf si vous souhaitez mapper ou non une classe à plusieurs unités de persistance.

L'application de gestion de portefeuilles d'actions utilise deux bases de données : les transactions doivent être enregistrées dans une base de données à part afin d'être utilisées dans d'autres applications annexes. La figure 6.1 présente les unités de persistance de l'application.

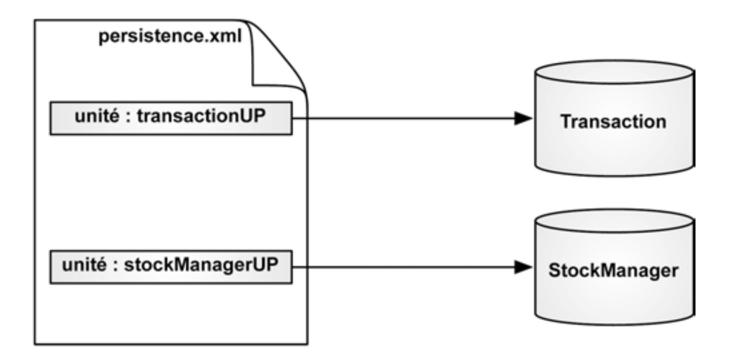


Figure 6.1 — Persistence.xml du portefeuille d'actions

Voici le fichier « persistence.xml », paramétré avec Hibernate, utilisé dans notre exemple :

```
<?xml version="1.0"?>
<persistence version="1.0">
  <persistence-unit name="transactionUP">
    <jta-data-source>java:StockTransactionDS</jta-data-source>
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
```

```
    <class>com.labosun.stockmanager.entity.Transaction</class>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop" />
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQLInnoDBDialect" />
    </properties>
    <exclude-unlisted-classes />
  </persistence-unit>

  <persistence-unit name="stockmanagerUP">
    <jta-data-source>java:StockMainDS</jta-data-source>
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>com.labosun.stockmanager.entity.FinancialProduct</class>
    <class>com.labosun.stockmanager.entity.Bond</class>
    <class>com.labosun.stockmanager.entity.Portfolio</class>
    <class>com.labosun.stockmanager.entity.Stock</class>
    <class>com.labosun.stockmanager.entity.Portfolio</class>
    <class>com.labosun.stockmanager.entity.User</class>
    <class>com.labosun.stockmanager.entity.AccountInfo</class>
    <class>com.labosun.stockmanager.entity.Address</class>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop" />
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQLInnoDBDialect" />
    </properties>
    <exclude-unlisted-classes />
  </persistence-unit>
</persistence>
```

Nous déclarons deux unités de persistance dans cet exemple (`transactionUP` et `stockmanagerUP`). Dans chacune, nous définissons les classes d'Entity Beans appartenant à l'unité et nous indiquons que les classes non déclarées sont exclues (*via* la balise : `<exclude-unlisted-classes>`).

Chaque unité de persistance utilise sa propre base de données. L'unité `transactionUP` est reliée à « StockMainDS » et `stockmanagerUP` est reliée à « StockTransactionDS ». Ces deux sources de données devant supporter les transactions, nous utilisons donc la balise `<jta-data-source>`.

Étant donné que ces unités de persistance utilisent Hibernate, nous devons paramétrer le moteur de persistance avec les propriétés disponibles (liées à Hibernate). Nous devons donc préciser la propriété `hibernate.hbm2ddl.auto` avec la valeur `update` (mettre à jour). Celle-ci permet de générer automatiquement le schéma de la base de données mais également de le mettre à jour en cas de modification de la structure des Entity Beans. Le schéma sera généré en fonction des annotations lues dans les classes d'entités.

La propriété `hibernate.dialect` paramètre le dialecte utilisé par Hibernate pour communiquer avec la base de données (ici MySQL).

Remarque : les propriétés disponibles sont expliquées en détail dans la documentation du fournisseur de persistance (pour Hibernate : www.hibernate.org).

6.3.2 Environnement Java EE

En environnement Java EE, le déploiement des classes d'EJB ainsi que de l'unité de persistance doit respecter certaines contraintes, selon le type d'utilisation. Vous devez donc respecter les règles suivantes.

Si l'unité de persistance doit être assemblée :

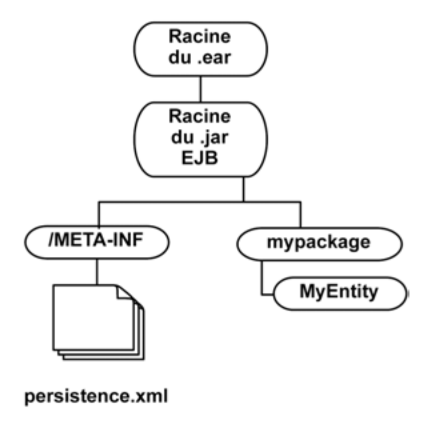


Figure 6.2 — Hiérarchie de l'archive EJB « .jar »

- En tant qu'ensemble de classes dans une archive EJB « .jar » (fig. 6.2) :
  – le fichier persistence.xml doit se trouver dans le répertoire « META-INF » ;
  – les classes doivent être à la racine du fichier JAR.

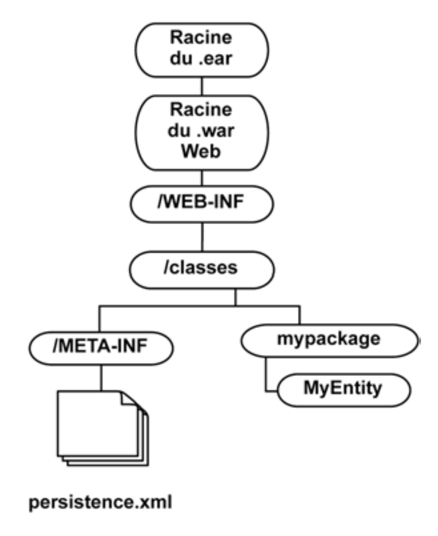


Figure 6.3 — Hiérarchie de l'archive web « .war »

- En tant qu'ensemble de classes dans une archive web .war (fig. 6.3) :
  – le fichier persistence.xml doit se trouver dans le répertoire « /WEB-INF/classes/META-INF » ;

– les classes doivent être dans le répertoire « /WEB-INF/classes » du fichier WAR.

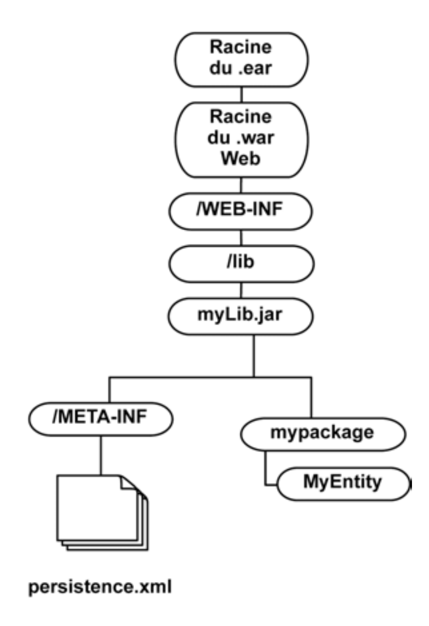


Figure 6.4 — Hiérarchie de l'archive web « .war » (EJB en tant que librairie)

- En tant qu'archive JAR externe incluse dans une archive WAR (fig. 6.4) : celle-ci doit se trouver dans le dossier « WEB-INF/lib ».

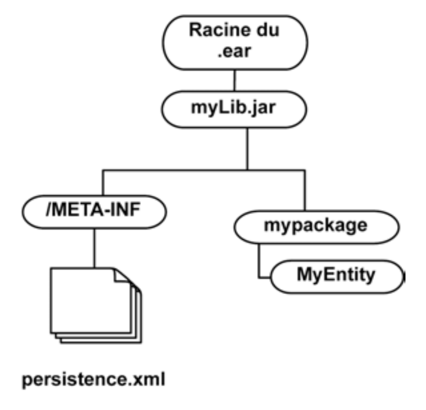


Figure 6.5 — Hiérarchie de l'archive JAR « .jar » (EJB en tant que librairie)

- En tant qu'archive JAR externe incluse dans une archive EAR (fig. 6.5) : celle-ci doit se trouver à la racine du fichier EAR.

Lorsqu'une contrainte n'est pas respectée, le conteneur ne démarre pas le moteur de persistance.

6.3.3 Environnement Java SE

Peu de différences existent en terme de packaging, entre une application Java SE et une application Java EE. Les classes Entity Bean doivent se trouver à la racine du fichier jar et le fichier « persistence.xml » doit être placé dans le dossier « META-INF ».

Toutefois, aucune source de données (*datasource*) ne peut être définie au sein d'une application Java SE. De ce fait, la configuration de cette source de données doit se faire par le biais des propriétés du fournisseur de persistance (*provider*). La différence majeure se situe dans le code de l'application. Le moteur de persistance étant lancé par le conteneur pour une application Java EE, ce lancement est à votre charge dans une application Java SE.

Voici le fichier persistence.xml associé, utilisant TopLink comme fournisseur de persistance :

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
  <persistence-unit name="contactUP" transaction-type="RESOURCE_LOCAL">
    <provider>oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider
    </provider>
    <class>com.labosun.contactbook.entity.Contact</class>
    <properties>
      <property name="jdbc.connection.string"
        value="jdbc:derby://localhost:1527/basededonnees"/>
      <property name="jdbc.driver"
        value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="jdbc.user" value="login"/>
      <property name="jdbc.password" value="password" />
      <property name="ddl-generation" value="dropandcreate"/>
    </properties>
  </persistence-unit>
</persistence>
```

Dans cet exemple, aucune source de données n'est spécifiée *via* la balise `<jta-datasource>`. On y précise cependant les paramètres de la connexion par les propriétés du fournisseur de persistance, qui en sera responsable.

Voici le même exemple utilisant Hibernate :

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
   <persistence-unit name="contactUP" transaction-type="RESOURCE_LOCAL">
    <class>com.labosun.contactbook.entity.Contact</class>
      <properties>
         <property name="hibernate.dialect"
           value="org.hibernate.dialect.MySQLInnoDBDialect"/>
         <property name="hibernate.connection.driver_class"
           value="com.mysql.jdbc.Driver"/>
         <property name="hibernate.connection.username" value="login"/>
         <property name="hibernate.connection.password" value="password"/>
         <property name="hibernate.connection.url"
           value="jdbc:mysql://localhost/basesdedonnees"/>
         <property name="hibernate.max_fetch_depth" value="3"/>
         <property name="hibernate.hbm2ddl.auto" value="update" />
      </properties>
```

```
        <exclude-unlisted-classes />
    </persistence-unit>
</persistence>
```

Conseil : vous pouvez paramétrer, de la même manière, la connexion à la base de données, dans une application Java EE. Néanmoins cette solution est déconseillée car elle n'est pas optimisée pour ce genre d'application. Il est préférable d'utiliser les services du conteneur tant que possible (datasource).

6.4 CYCLE DE VIE DU CONTEXTE DE PERSISTANCE

Tout comme les Entity Beans, le contexte de persistance représenté par l'Entity Manager est lui aussi soumis à une durée de vie. Il existe deux comportements différents, pour les instances d'Entity Manager, qui dépendent du Session Bean utilisé :

- « transaction-scoped persistence context » : durée de vie liée à une seule transaction.
- « extended persistence contexts » : durée de vie liée à celle d'un Stateful Session Bean (et donc plusieurs transactions).

Remarque : dans un environnement Java SE, seul « extended persistence contexts » est disponible.

6.4.1 Transaction-scoped persistence context

Lorsqu'un Entity Manager est de type « transaction-scoped », le cycle de vie est automatiquement géré par le conteneur Java EE. Cela induit une totale transparence pour l'application.

Dans le cas d'un « transaction-scoped persistence context », le contexte de persistance s'initialise lorsque l'application demande un Entity Manager au sein d'une transaction déjà active (fig. 6.6). Le contexte persistant s'arrête quand la transaction associée est validée ou annulée. Si un Entity Manager est appelé en dehors de toute transaction, le contexte est créé et supprimé respectivement au début et à la fin de la méthode. Lorsque le contexte de persistance est détruit, l'ensemble des entités managées deviennent détachées.

```
@Stateless
public class WebClientServiceBean implements WebClientService {
  @PersistenceContext(unitName="stockManagerUP")
  EntityManager em;

  public User createUser(User user) {
    em.persist(user);
    user.setValid(false);
    return user;
  }
}
```

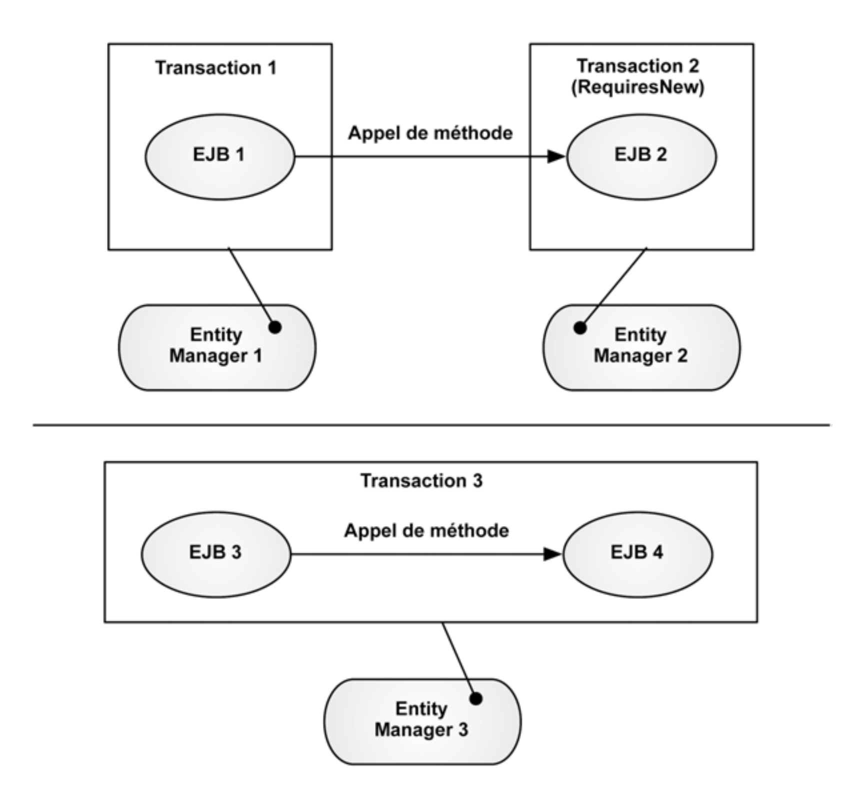


Figure 6.6 — Transaction-scoped persistence context

Dans l'exemple précédent, la méthode `createUser(User user)` enregistre un nouvel utilisateur. Même si la méthode `persist(Object o)` est appelée avant la modification de la propriété `valid` de l'instance `user`, la modification sera prise en compte et synchronisée en base de données car `user` est *managée*. Lorsque la transaction se termine, `user` est alors détaché et les modifications apportées ne sont plus synchronisées avec la base de données.

6.4.2 Extended persistence context

Il est également possible de définir un contexte de persistance avec une durée de vie s'étalant sur plusieurs transactions. On parle alors de « extended persistence context ». Chaque instance d'Entity Bean *managée* par ce type de contexte le reste même après la fin d'une transaction. Les instances sont détachées à la fermeture du contexte (fig. 6.7).

Dans ce cas, la durée de vie d'un contexte de persistance étendu doit être gérée par l'application (de la création jusqu'à la suppression). Toutefois, cette gestion peut être automatiquement gérée avec un Stateful Session Bean.

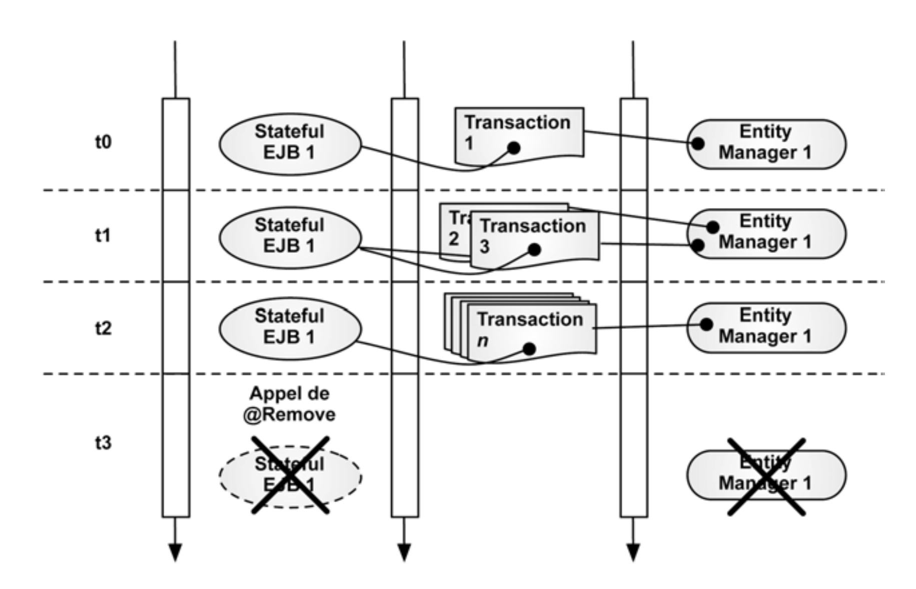


Figure 6.7 — Extended persitence context

Ce type de contexte de persistance n'a d'utilité que s'il est utilisé au sein d'un Stateful Session Bean. En effet, c'est le seul composant à garder un état conversationnel avec le client entre plusieurs appels de méthodes. Avec les Stateless Session Beans ou les Message Driven Beans il est impossible de mettre en place un cache local, ceux-ci n'étant pas affectés à un client particulier.

Les transactions sont cependant de type « method scoped » (voir chapitre 9), c'est-à-dire qu'elles démarrent et finissent suivant la portée de la méthode. De ce fait, les objets attachés durant l'exécution de celle-ci sont détachés aussitôt à la fin de la méthode. Dans le cas du Stateful Session Bean, le contexte de persistance étendu est créé en même temps que le composant et fermé automatiquement à la suppression de l'instance du Session Bean. Vous pouvez donc garder des entités *managées* en tant que variable d'instance.

Voici une comparaison sur l'utilisation du contexte de persistance avec état (Stateful) et sans état (Stateless).

```
@Stateful
public class WebClientServiceBean implements WebClientService {
  @PersistenceContext(unitName="stockManagerUP",
    type=PersistenceContextType.EXTENDED)
  EntityManager em;

  private User currentUser;

  public User loadUser(int idUser) {
    currentUser = em.find(User.class, idUser);
    return currentUser;
  }
  public void updateUser(User updatedUser) {
```

```
      currentUser.setAddress(updatedUser.getAddress());
      currentUser.setFirstName(updatedUser.getFirstName());
    }
  }
```

La méthode loadUser() charge l'utilisateur courant et le retourne. La méthode updateUser() permet de modifier les données de l'utilisateur.

Vous pouvez remarquer qu'aucune méthode de l'Entity Manager n'est appelée pour mettre à jour l'entité currentUser. Cela est dû au fait que cette entité est *managée* et le reste tout au long de la vie du Stateful Session Bean.

```
@Stateless
public class WebClientServiceBean implements WebClientService {
  @PersistenceContext(unitName="stockManagerUP")
  EntityManager em;

  public User loadUser(int idUser) {
    User currentUser = em.find(User.class, idUser);
    return currentUser;
  }
  public void updateUser(User updatedUser) {
    User currentUser = this.loadUser(updatedUser.getId());
    currentUser.setAddress(updatedUser.getAddress());
    currentUser.setFirstName(updatedUser.getFirstName());
  }
}
```

La version Stateless est moins optimisée. Un contexte de persistance est créé puis détruit à chaque appel de méthodes. Les entités chargées sont alors détachées entre chaque appel de méthodes. Toutefois, la consommation mémoire est moindre qu'avec le Stateful Session Bean car les états ne sont pas conservés.

6.5 LA PERSISTANCE *VIA* L'ENTITY MANAGER

Lors du développement d'une application Java EE, il est possible que certaines instances ne soient pas utilisées de la même manière que les autres. Il est possible que la majorité de celles-ci doivent être rendues persistantes dans la source de données, mais que certaines doivent rester temporaires (en mémoire).

Le conteneur ne peut donc pas savoir si telle ou telle instance doit être liée à la source de données. C'est le développeur qui décide de sauvegarder, modifier, supprimer… ces instances. Pour cela, l'API « Persistence 1.0 » fournit l'interface EntityManager qui permet d'effectuer ces opérations d'accès aux données. Toutefois, pour créer un objet de ce type, il faut utiliser la fabrique associée : EntityManagerFactory.

Remarque : l'accès aux données est représenté par les opérations d'ajout, lecture, modification et suppression (défini par l'acronyme CRUD : *Create Read Update Delete*). On implémente généralement le Pattern DAO pour uniformiser ces accès.

L'Entity Manager étant une interface, la classe d'implémentation est différente suivant le fournisseur utilisé (défini dans le fichier « persistence.xml » avec la balise `<provider>`).

Par défaut, c'est le conteneur qui instanciera l'Entity Manager et qui gérera son cycle de vie. On parle alors de gestionnaire d'entités géré par le conteneur (*container-managed entity manager*).

Il vous est cependant possible de gérer manuellement ce cycle de vie. On parle alors de gestionnaire d'entité géré par l'application (*application-managed entity manager*). Ce sera le cas des applications Java SE, où seuls un Entity Manager géré par l'application est disponible.

6.5.1 Obtenir un Entity Manager

Il existe plusieurs manières d'obtenir un objet `EntityManager` qui ont chacune des spécificités quant à leur utilisation :

- Injection par annotation : utilisée exclusivement en environnement Java EE.
- Utilisation de la fabrique `EntityManagerFactory` : utilisée pour gérer de façon manuelle la création d'`EntityManager`. Elle est obligatoire pour une application Java SE.

Dans la plupart des cas, la manipulation des données *via* un conteneur EJB se réalise en définissant un Session Bean en « façade ».

Remarque : le *design pattern* « façade » fournit une interface d'accès uniforme à un ensemble de sous-systèmes. L'application utilise donc un ensemble de systèmes sans réellement en avoir conscience et sans avoir les difficultés d'utilisation que chaque système peut induire.

Ce Session Bean accède donc au contexte de persistance et peut ensuite travailler avec les instances des Entity Beans, *via* l'Entity Manager. C'est ce qui sera présenté dans les exemples des parties suivantes.

L'interface EntityManagerFactory

Nous avons vu précédemment que la création d'instance d'Entity Manager passe par l'utilisation de l'Entity Manager Factory. Cette interface propose plusieurs méthodes de création :

- `EntityManager createEntityManager()` : retourne une instance de `EntityManager` de type « extended ».
- `EntityManager createEntityManager(Map properties)` : retourne une instance de `EntityManager` en utilisant les propriétés passées en paramètre. Ces propriétés permettent de surdéfinir celles du fichier « persistence.xml » ou de les étendre.

- `void close()` : ferme la `EntityManagerFactory`, permettant de libérer les ressources qu'elle utilise.
- `boolean isOpen()` : permet de connaître si l'`EntityManagerFactory` est toujours valide.

En environnement Java EE

Comme nous venons de le voir, vous pouvez utiliser l'interface `EntityManagerFactory` pour créer et gérer manuellement les instances d'`EntityManager`. Toutefois, nous verrons qu'il est possible de laisser le conteneur se charger d'injecter une instance d'un Entity Manager lors de l'instanciation du Session Bean. Cette méthode est sans doute la plus simple et la plus pratique.

Afin d'indiquer au conteneur que le Bean est dépendant d'un `EntityManagerFactory` ou d'un `EntityManager`, vous devez annoter la variable d'instance de type `EntityManagerFactory` ou `EntityManager` avec `@PersistenceContext`. Cette annotation admet plusieurs attributs :

- `name` : déclare un nom local référencé sur une unité de persistance déployée (référence locale vers une unité de persistance).
- `unitName` : définit le nom de l'unité de persistance à utiliser pour injecter l'`EntityManager`.
- `type` : indique le type contexte de persistance injecté (`EXTENDED` ou `TRANSACTION` par défaut).

Voici un exemple d'utilisation au sein de l'application de gestion de portefeuille d'actions. Ici, un `EntityManagerFactory`, lié à l'unité de persistance nommée « stockmanagerUP », est injecté.

```
@Stateless
@Local({CommonService.class})
public class CommonServiceBean implements CommonService {

  @PersistenceContext(unitName="stockmanagerUP")
  protected EntityManagerFactory emStockManagerFactory;
  //...
}
```

Si vous utilisez l'`EntityManagerFactory`, vous devez explicitement créer votre `EntityManager`. Cette instance sera de type `EXTENDED`. Vous devrez donc appeler la méthode `EntityManager.joinTransaction()` si l'unité de persistance est configurée avec le type de transaction JTA (par défaut en environnement Java EE). Dans le cas contraire, des informations pourraient être perdues, étant donné qu'aucune synchronisation ne sera faite avec la base de données.

```
//...
  @PersistenceContext(unitName="stockmanagerUP")
  protected EntityManagerFactory emStockManagerFactory;

  public void addUser(User user) {
```

```
    EntityManager em = emStockManagerFactory.createEntityManager();
    em.jointTransaction();
    //...
  }
```

Comme vous pouvez le constater, l'utilisation de l'EntityManagerFactory est une charge pour les développeurs. Il est, cependant, possible de laisser cette charge au conteneur en lui demandant d'injecter directement une instance EntityManager.

```
@Stateless
@Local({CommonService.class})
public class CommonServiceBean implements CommonService {

  @PersistenceContext(unitName="stockmanagerUP")
  protected EntityManager emStockManager;
  //...
}
```

Dans cet exemple, l'EntityManager injecté est automatiquement associé à la transaction courante jusqu'à la fin de celle-ci. Nous sommes alors en présence d'un « transaction-scoped persistence context ». Le même contexte de persistance est alors utilisé tant que vous travaillez dans la transaction de départ. Il devient alors inutile d'appeler la méthode close() de l'EntityManager, son cycle de vie étant automatiquement géré par le conteneur (en cas d'appel de cette méthode, une exception IllegalStateException est levée).

L'injection d'un *extended persistence context* ne prend de sens que dans un Stateful Session Bean. En effet, la durée de vie de celui-ci a réellement un intérêt dans l'application.

Pour spécifier le fait d'utiliser un contexte de persistance de type étendu, vous devez utiliser l'attribut type de l'annotation @PersistenceUnit.

```
@Stateful
@Local({RichClientService.class})
public class RichClientServiceBean implements RichClientService {

  @PersistenceContext(unitName="stockmanagerUP",
    type=PersistenceContextType.EXTENDED)
  protected EntityManager emStockManager;
  //...
}
```

Le contexte de persistance est créé au même moment que le Session Bean et a le même cycle de vie que celui-ci. Il est fermé lorsque le Session Bean est détruit. Durant toute la vie du Session Bean, les instances des Entity Beans associées au contexte de persistance sont *managées*.

En environnement Java SE

Si l'application s'exécute dans un environnement Java SE, l'initialisation du conteneur est à la charge du développeur. La classe javax.persistence.Persistence sert à cette initialisation et fournit les méthodes d'accès à l'EntityManagerFactory.

- `EntityManagerFactory createEntityManagerFactory(String unitName)` : créé un `EntityManagerFactory` lié au contexte persistant dont le nom est passé en paramètre.
- `EntityManagerFactory createEntityManagerFactory(String unitName, Map properties)` : créé un `EntityManagerFactory` et spécifie des propriétés spéciales permettant de surdéfinir ou d'étendre les propriétés définies dans le fichier « persistence.xml ».

Voici un exemple de création d'un `EntityManager` en environnement Java SE.

```
public class PersistenceLauncher {
  public static void main (String[] args) {
    EntityManagerFactory emf =
      Persistence.createEntityManagerFactory("stockManagerUP");
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    //...
    // opérations sur les entités
    //...
    em.getTransaction().commit();
    em.close();
    emf.close();
  }
}
```

La première ligne de la méthode `main()` initialise le moteur de persistance en chargeant le fichier « persistence.xml ». La seconde crée une instance d'`EntityManager` afin de pouvoir travailler avec les Entity Beans.

La gestion des transactions est manuelle. Vous devez donc définir le début et la fin pour chaque transaction effectuée.

Attention : lorsque vous avez terminé vos opérations, n'oubliez pas d'appeler la méthode `EntityManagerFactory.close()` afin de libérer les ressources associées au contexte de persistance.

Resource Local Transactions

En environnement Java EE, le contexte de persistance utilise généralement la transaction JTA associée par le conteneur. Cependant dans un environnement Java SE (non Java EE) cela n'est pas le cas !

Il faut alors gérer les transactions de façon manuelle. Pour cela, l'API « Persistence 1.0 » fournit l'interface `EntityTransaction`. Un objet de ce type est récupéré *via* l'appel de la méthode `EntityManager.getTransaction()`.

Remarque : la gestion des transactions avec cette API semblera très similaire à l'API JDBC.

Cette interface fournit un ensemble de méthodes liées à la gestion des transactions :

- `void begin()` : démarre une nouvelle transaction. Cette méthode lève une `IllegaleStateException` si une transaction est déjà active.
- `void commit()` : valide les opérations de la transaction courante. Cette méthode lève une `IllegaleStateException` si aucune transaction n'est active.
- `void rollback()` : annule les opérations de la transaction courante. Cette méthode lève une `IllegaleStateException` si aucune transaction n'est active.
- `boolean isActive()` : retourne `true` si la transaction courante est active et `false` dans l'autre cas.

Ce type de gestion a spécialement été conçu pour les environnements n'ayant pas accès facilement à JTA, en l'occurrence, Java SE. Voici un exemple d'utilisation d'`EntityTransaction` dans une application Java *standalone*.

```
public class Launcher {

  public static void main(String[] args) {
    // Utilise la configuration persistence.xml
    EntityManagerFactory emf =
      Persistence.createEntityManagerFactory("contactUP");
    // Retrieve an application managed entity manager
    EntityManager em = emf.createEntityManager();

    // Crée une instance non managée de Contact
    Contact contact = new Contact();
    contact.setFirstName("Cyril");
    contact.setLastName("JOUI");
    // démarre la transaction
    em.getTransaction().begin();

    // opérations dans la transaction
    em.persist(contact);
    //...

    // valide la transaction
    em.getTransaction().commit();

    // ferme le contexte de persistance
    em.close();
    emf.close();
  }
}
```

6.5.2 Travailler avec l'Entity Manager

L'interface `javax.persistence.EntityManager` est le point central de la gestion des entités persistantes. Elle offre des méthodes d'ajout, modification, suppression, recherche et un accès à l'API Query (voir chapitre 7).

Nous allons détailler les différentes méthodes de cette interface et expliquer leur comportement.

Enregistrer les entités

Enregistrer une entité consiste à l'insérer dans la base de données. Cette opération se traduit par l'appel de la méthode `EntityManager.persist(Object o)`. Celle-ci ne s'applique que sur des entités encore non enregistrées dans la base de données (fig. 6.8).

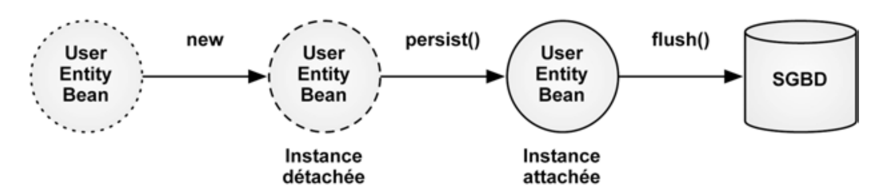


Figure 6.8 — Principe de la méthode `persist()`

Pour enregistrer une entité, vous devez tout d'abord créer une instance, puis affecter ses propriétés et ses relations avec les valeurs souhaitées, comme vous pouvez le faire avec un objet Java. Finalement, il vous suffit d'appeler la méthode `persist()`.

```
User newUser = new User();
// affecte les relations
newUser.setFirstname("Cyril");
newUser.setEmail("popom@supinfo.com");
//...
entityManager.persist(newUser);
```

Une fois cette méthode appelée, l'instance `newUser` devient *managée* et son insertion en base de données est mise dans la file d'attente de l'Entity Manager. Si la méthode `persist()` est appelée dans une transaction, l'insertion en base de données peut être faite immédiatement ou mise en file d'attente jusqu'à ce que la transaction se termine. Cela dépend du mode utilisé pour la synchronisation à la base de données.

Il n'est possible d'appeler la méthode `persist()` en dehors d'une transaction seulement si l'Entity Manager est de type étendu (`EXTENDED`). Dans ce cas-là, l'insertion est mise en file d'attente jusqu'à ce que le contexte soit associé à une transaction.

> **Remarque :** pour rappel, si vous injectez un contexte persistant étendu (*via* `@PersistenceContext`) alors il est automatiquement associé à la transaction. Dans les autres cas (manuellement *via* `EntityManagerFactory`) vous devez appeler la méthode `EntityManager.joinTransaction()` pour associer le contexte à la transaction courante.

Si vous appelez la méthode `persist()` avec un type d'argument autre qu'un Entity Bean, une `IllegaleArgumentException` sera levée. De même, si vous appelez

cette méthode *via* un « transaction-scoped persistence context » et qu'aucune transaction n'existe, une `TransactionRequiredException` sera levée.

Retrouver les entités

Une fois les objets sauvegardés, il est important de pouvoir les récupérer. Il existe deux façons de récupérer ces objets de la base de données. Nous allons détailler la récupération d'objets à partir de leur clé primaire. L'autre solution consiste à travailler avec les requêtes EJB-QL (voir chapitre 7).

L'Entity Manager a deux méthodes simples pour trouver une entité à partir de sa clé primaire.

- `<T> T find(Class<T> entity, Object primaryKey)`
- `<T> T getReference(Class<T> entity, Object primaryKey)`

Ces deux méthodes se ressemblent étroitement et prennent toutes les deux les mêmes paramètres : la classe de l'entité et l'instance de la clé primaire. L'utilisation des génériques évite d'avoir à *caster* la valeur de retour. Toutefois des différences existent.

La méthode `find()` retourne `null` si aucune entité n'est associée à la clé primaire demandée. Elle initialise également les états de base du *lazy-loading* associés aux propriétés.

```
User user1 = entityManager.find(User.class, 1);
if(user1 == null) { ... }
```

La méthode `getReference()` se comporte différemment si l'entité n'est pas trouvée en base de données. Dans ce cas-là, elle lance une `javax.persistence.EntityNotFoundException`. De plus, elle ne garantit pas l'initialisation des états de l'entité.

Modifier les entités

Il existe deux façons de modifier une entité. Soit vous la chargez depuis la base de données et vous lui appliquez les modifications au sein de la transaction, soit vous souhaitez mettre à jour un objet détaché et dans ce cas-là vous devez utiliser la méthode `merge()`.

Lorsque vous récupérez une entité *via* `find()`, `getReference()` ou par l'intermédiaire d'une requête, celle-ci se retrouve *managée* jusqu'à la fermeture du contexte de persistance. Vous pouvez alors changer les propriétés de cette instance, les modifications seront synchronisées automatiquement.

```
public void changeUserEmail(int userId, String email) {
  User currentUser = entityManager.find(User.class, userId);
  currentUser.setEmail(email);
}
```

Dans l'exemple précédent, l'Entity Manager doit être associée à la transaction courante si l'on souhaite que les modifications soient enregistrées. Celles-ci sont

réellement affectées en base de données lorsque la transaction se termine ou que la méthode `flush()` est appelée explicitement (fig. 6.9).

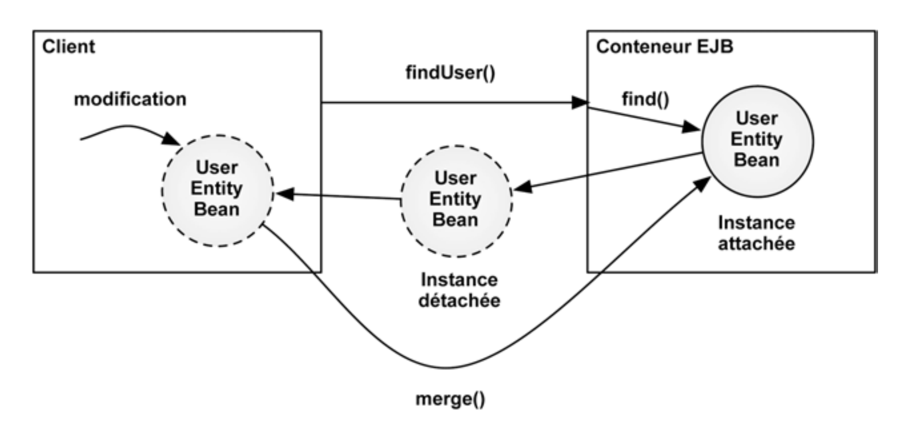


Figure 6.9 — Principe de la méthode `merge()`

L'autre solution est à utiliser lorsque vous souhaitez fusionner les modifications faites sur un objet détaché vers le contexte de persistance. C'est le cas, par exemple, lorsqu'une entité quitte le conteneur EJB vers une autre application (Web ou client riche). Vous devez alors utiliser la méthode `EntityManager.merge()` afin de rattacher l'instance au contexte de persistance. Imaginons qu'un client riche appelle la méthode `findUser()` suivante :

```
@PersistenceContext
EntityManager em;

public User findUser(int userId) {
  return em.find(User.class, userId);
}
```

Le contexte de persistance se ferme après l'exécution de la méthode `findUser()` car il est de type « transaction-scoped » (nous supposons que le Session Bean utilise la gestion par défaut des transactions). À partir de là, l'instance retournée est détachée et les modifications appliquées ne sont plus synchronisées. Si l'application cliente souhaite enregistrer les modifications faites en local, elle doit renvoyer l'instance au Session Bean.

```
User returnedUser = adminService.findUser(2);
returnedUser.setEmail("nouvelemail@supinfo.com");
adminService.updateUser(returnedUser);
```

Dans ce cas-là, la méthode `updateUser()` doit utiliser `EntityManager.merge()` afin de rattacher l'instance au contexte.

```
@PersistenceContext
EntityManager em;

public void updateUser(User user) {
```

```
  User attachedUser = em.merge(user);
}
```

Différents comportements s'appliquent suivant le contexte dans lequel vous appelez la méthode merge().

Si l'Entity Manager ne contient pas d'instance de l'entité avec la même clé primaire, alors il crée une copie complète de l'objet passé en argument et la retourne. Cette copie est alors attachée au contexte de persistance et les modifications faites dessus sont synchronisées avec la base de données.

Si l'Entity Manager détient déjà une instance de l'entité ayant la même clé primaire, alors le contenu du paramètre (ici user) est copié dans l'instance attachée.

Dans les deux cas, le paramètre user de la méthode updateUser() reste détaché et donc n'est pas synchronisé avec la base de données. C'est la raison pour laquelle le développeur doit travailler avec l'instance retournée par la méthode merge() s'il doit faire d'autres modifications sur l'entité.

Supprimer les entités

La méthode EntityManager.remove() sert à demander la suppression d'une entité de la base de données (fig. 6.10). Nous parlons de « demande » car la suppression n'est pas effective immédiatement mais seulement à l'appel de la méthode flush() ou à la fermeture du contexte de persistance. Entre-temps, il est possible d'annuler la suppression.

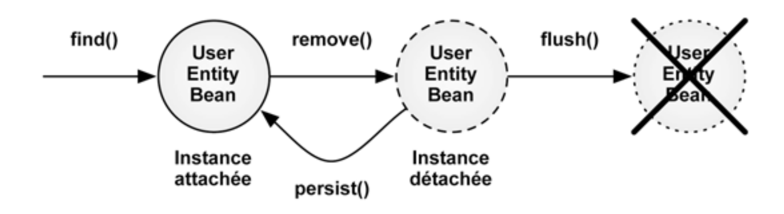


Figure 6.10 — Principe de la méthode remove()

```
@PersistenceContext
EntityManager em;


public void removeUser(int userId) {
  User userToRemove = em.find(userId);
  em.remove(userToRemove);
}
```

L'appel à la méthode remove() détache l'entité du contexte de persistance. Pour annuler cette suppression (dans le même contexte de persistance) il faut appeler la méthode persist() afin de rattacher l'instance au contexte.

Recharger les entités

Si vous savez que l'instance d'une entité ne reflète pas les valeurs de la base de données (parce que celle-ci a été modifiée entre-temps...) vous pouvez utiliser la méthode `EntityManager.refresh()` afin de recharger l'entité depuis la base de données. Cette opération écrase toutes les modifications qui ont pu être apportées à l'entité.

```
@PersistenceContext
EntityManager em;

public void workOnUser(int userId) {
  User user = em.find(userId);
  user.setEmail("nomail@supinfo.com");
  // recharge l'objet depuis la base de données
  em.refresh(user);
}
```

Dans l'exemple précédent, l'email n'est pas modifié à cause de l'appel à la méthode `refresh()`.

Flush

Lorsque vous appelez les méthodes `persist(), merge() ou remove()`, les changements ne sont pas synchronisés immédiatement. Cette synchronisation s'établit automatiquement à la fin d'une transaction ou lorsque l'Entity Manager décide de vider (*to flush*, en anglais) sa file d'attente. Toutefois, le développeur peut forcer la synchronisation en appelant explicitement la méthode `flush()` de l'Entity Manager. Vous pouvez également définir le comportement que doit avoir l'Entity Manager pour effectuer automatiquement ses appels à `flush()`, *via* la méthode `setFlushMode()`. Cette méthode prend en paramètre un objet `FlushModeType`, une énumération regroupant deux valeurs :

- `AUTO` : définit le comportement par défaut (présenté tout au long de cette partie).
- `COMMIT` : spécifie que les changements sont envoyés à la base de données seulement lorsque la transaction est *commitée* et non après chaque requête.

> **Attention** : `flush()` et `commit()` sont deux opérations différentes ! `flush()` envoie les requêtes SQL aux sources de données alors que `commit()` valide la transaction au niveau de ces sources.

Le mode `AUTO` est utilisé dans la plupart des cas : il permet de maintenir une cohérence des données. Par exemple, lorsque vous faites une requête sur la base de données, vous souhaitez que toutes les modifications faites avant aient été envoyées. Dans le cas contraire, vous risquez de ne pas avoir les toutes dernières informations dans les résultats de la requête. Bien entendu, vous souhaitez également que les

modifications demandées soient exécutées lorsque la transaction est *commitée*. L'inconvénient du mode `AUTO` est qu'il n'est pas de tout repos pour la base de données car de nombreux allers et retours sont exécutés.

Le mode `COMMIT` est utilisé lorsqu'on souhaite optimiser les performances de certaines transactions. L'utilisation de ce mode permet de réunir l'ensemble des requêtes à exécuter et à les transmettre d'un seul coup à la base de données. Cela provoque un avantage considérable en temps d'exécution. Sachant que l'exécution d'une requête UPDATE oblige de verrouiller certaines lignes, il est avantageux de les regrouper. De même, cela évite d'exécuter plusieurs *commit* qui peuvent être gourmand en ressources.

Il est apparemment plus facile d'utiliser le mode `AUTO`, celui-ci gérant de manière cohérente les données. Toutefois, il peut être intéressant d'utiliser le mode `COMMIT` lorsque vous souhaitez optimiser votre application et la rendre plus performante.

Autres opérations

D'autres opérations sont disponibles dans l'Entity Manager :

- `boolean contains(Object entity)` : retourne `true` si l'entité passée en paramètre est attachée au contexte persistant courant.
- `void clear()` : permet de détacher l'ensemble des entités liées au contexte de persistance courant.

> **Attention** : toutes les modifications affectées sont perdues à l'appel de la méthode `clear()`. Pour éviter cela, il faut appeler la méthode `flush()` avant l'appel de `clear()`.

6.6 CYCLE DE VIE D'UN ENTITY BEAN

Comme en EJB 2, le cycle de vie d'un Entity Bean peut être géré. La gestion de celui-ci vous permet d'effectuer automatiquement des opérations liées à l'enregistrement, la modification, la suppression, ou encore le chargement d'un Bean.

En EJB 3, il est possible d'utiliser, pour la gestion du cycle de vie, soit une méthode de la classe `EntityBean`, soit une classe externe `Listener`. Dans les deux cas, l'appel des méthodes liées au cycle de vie est réalisé par le conteneur. On parle de *callback methods* (méthodes de retour).

En parallèle avec EJB 2, on retrouve le principe des méthodes `ejbCreate`, `ejbPostCreate`, `ejbLoad`, `ejbStore`, `ejbRemove`. Toutefois de nombreuses améliorations et simplifications ont été mises en place.

6.6.1 États d'un Entity Bean

Le cycle de vie d'une instance d'un Entity Bean comprend quatre états distincts qu'il faut bien connaître et comprendre. Voici une description de ces différents états :

- *new* (nouveau) : signifie que l'instance n'est associée à aucun contexte persistant. Cet état résulte de l'instanciation de l'Entity Bean *via* `new`.
- *managed* (gérée) : signifie que l'instance possède une identité associée au contexte persistant. Une instance est généralement dans cet état lorsqu'elle vient d'être enregistrée, modifiée ou récupérée.
- *detached* (dissociée) : signifie que l'instance n'est plus associée au contexte persistant d'où elle provient. C'est généralement le cas lorsque l'instance est initialisée dans le conteneur puis envoyé à un autre tiers (présentation, web…).
- *removed* (supprimée) : signifie que l'instance a une identité associée à un contexte persistant mais qu'elle est destinée à être retirée de la base de données.

Un schéma récapitulatif des états et des changements d'état est disponible à la fin de cette partie (fig. 6.11).

6.6.2 Définition des « callback methods »

Ces méthodes, lorsqu'elles sont déclarées dans la classe du Bean, doivent avoir la forme suivante (où `<METHOD>` correspond au nom de la méthode) :

```
void <METHOD>()
```

Pour spécifier au conteneur à quel moment du cycle de vie doit être appelée cette méthode, vous devez l'annoter avec une annotation de cycle de vie. Celles-ci seront détaillées dans le chapitre suivant.

```
@Entity
public class User implements Serializable {
  ...
  @PostPersist
  public void alertNewUser() {
    // envoie d'un mail à l'administrateur
    //...
  }
}
```

Cet exemple définit la méthode `alertNewUser()` qui envoie un mail à l'administrateur lorsqu'un nouvel utilisateur est ajouté. L'annotation `@PostPersist` indique au conteneur d'appeler cette méthode après l'enregistrement de l'objet en base de données.

Si l'on souhaite utiliser une classe annexe, afin de ne pas surcharger la classe de l'Entity Bean, les méthodes de cette classe devront être de la forme suivante (où `Entity` est la classe de l'Entity Bean) :

```
void <METHOD>(Entity)
```

En voici un exemple :

```
public class UserAlert {
@PostPersist
public void alertNewUser(User user) {
  // envoie d'un mail à l'administrateur
  //...
}
```

Toutefois, il faut assigner cette classe *listener* à l'Entity Bean. Pour cela, il faut utiliser l'annotation `@EntityListeners` sur la classe de celui-ci. Cette annotation regroupe les différentes classes *listener*.

```
@EntityListeners({UserAlert.class})
public class User implements Serializable {
  //...
}
```

6.6.3 Annotations du cycle de vie

Il existe au total 7 annotations permettant de spécifier les différents moments du cycle de vie d'un Entity Bean liés aux méthodes : `persist()`, `remove()`, `merge()` et `find()` de l'Entity Manager.

Les méthodes annotées avec `@PrePersist` ou `@PreRemove` sont invoquées sur un Entity Bean **avant** l'exécution des méthodes `persist()` et `remove()` de l'Entity Manager.

Les méthodes annotées avec `@PostPersist` ou `@PostRemove` sont invoquées sur un Entity Bean **après** l'exécution des méthodes `persist()` et `remove()` de l'Entity Manager. Ces méthodes sont appelées après l'exécution des requêtes `INSERT` ou `DELETE`. Les valeurs des clés primaires auto-générées sont disponibles dans la méthode annotée par `@PostPersist`.

Les méthodes annotées avec `@PreUpdate` ou `@PostUpdate` sont invoquées sur un Entity Bean respectivement **avant** ou **après** la mise à jour de la base de données.

La méthode annotée avec `@PostLoad` est invoquée après le chargement, depuis la base de données, de l'Entity Bean dans le contexte de persistance courant ou après l'exécution de la méthode `refresh()`. Cette méthode est appelée avant qu'une requête retourne les résultats.

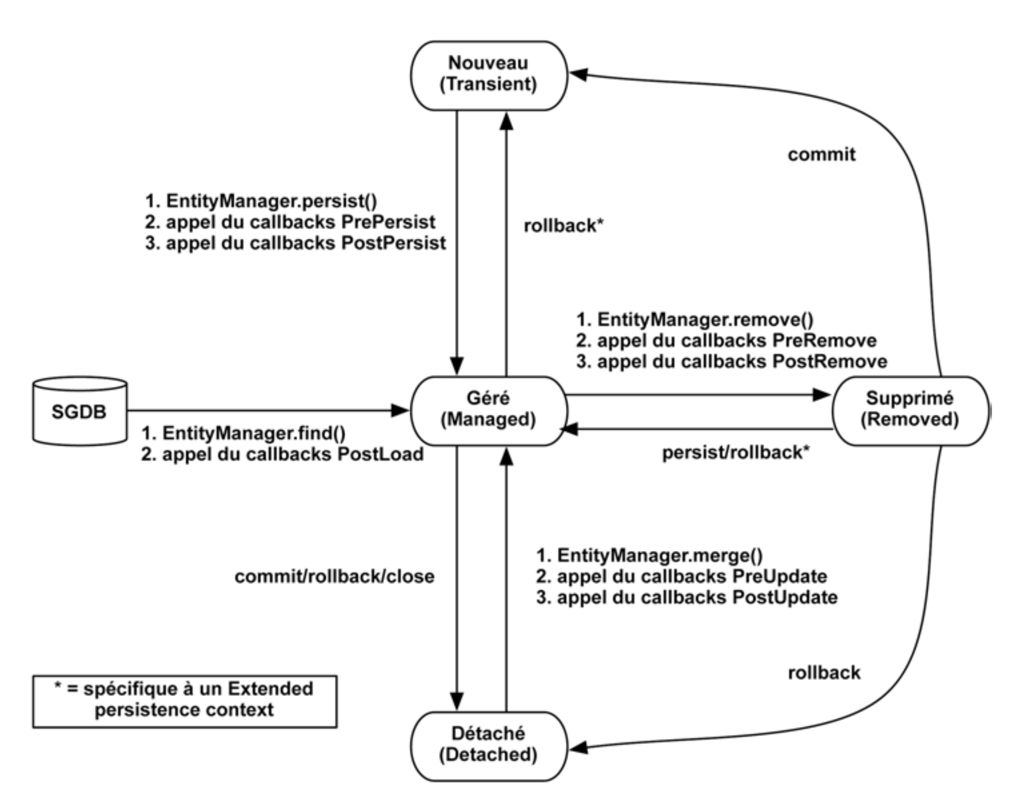


Figure 6.11 — Schéma récapitulatif du cycle de vie d'un Entity Bean

6.6.4 Principe du « lazy loading »

Comme nous l'avons vu dans le chapitre 5, les propriétés peuvent être chargées de façon *lazy* (paresseuse) lorsque le programme tente d'y accéder. Ce principe s'appelle le *lazy loading*. Même si cela semble intéressant, il est nécessaire de bien connaître son fonctionnement afin de ne pas se retrouver avec des erreurs de chargement.

Prenons l'exemple d'un utilisateur ayant plusieurs portefeuilles d'actions. Il existe un champ relationnel (une collection) `portfolios` dans `User`. Si ce champ est marqué pour un chargement de type `FetchType.Lazy` (par défaut pour les champs relationnels multi-valuées), alors il est chargé à la demande et fonctionne si cette instance est managée.

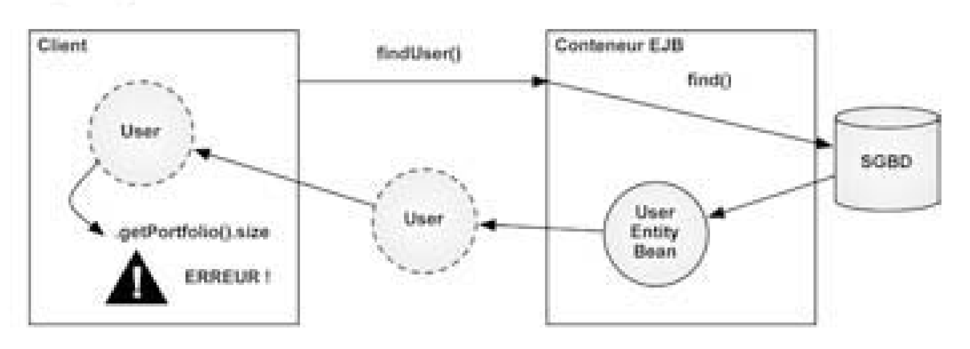
```
public class User {
//...

  // propriété par défaut en mode de chargement lazy
  @OneToMany
  public Collection<Portfolio> getPortfolios() {
    return portfolios;
  }
```

```
    @Basic(fetch=FetchType.LAZY)
    @Lob
    public String getCVContent() { return cvContent; }
}
```

Dans cet exemple, le chargement de la propriété `portfolios` est de type *lazy* automatiquement. Si une propriété persistante à besoin d'être chargée à la demande, il faut alors utiliser l'attribut `fetch` de l'annotation `@Basic` afin de forcer le type de chargement en *lazy*. Ici, la propriété `cvContent` représente le contenu du *curriculum vitae* de l'utilisateur et peut donc contenir de nombreux caractères (fig. 6.12).

Que se passe-t-il si une application cliente récupère une entité (qui est alors détachée) et tente d'accéder à un champ relationnel non initialisé ? La spécification n'est pas très claire là-dessus. Cependant, la plupart des fournisseurs de persistance lèvent une exception afin d'avertir que la propriété demandée n'a pas été initialisée.



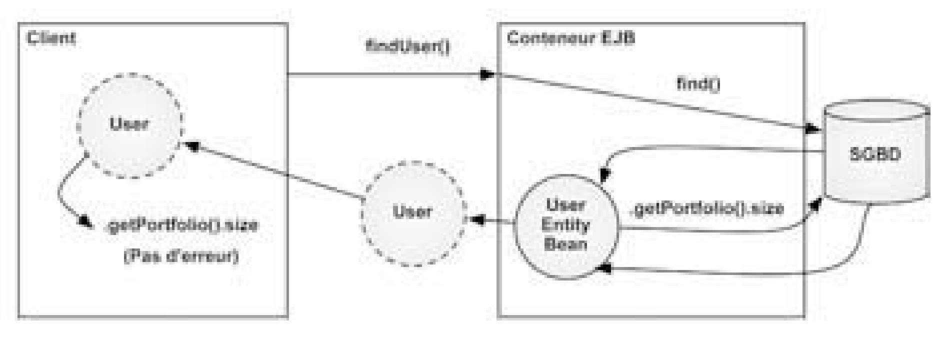


Figure 6.12 — Schéma d'explication du *lazy loading*

Pour pallier cela, il faut s'assurer que toutes les propriétés de l'entité utilisée chez le client sont initialisées avant que l'objet soit détaché du contexte de persistance.

Pour cela, il suffit de demander le chargement des données quand l'entité est encore *managée*.

```
User user = entityManager.find(User.class, userId) ;
// appeler une méthode de la collection pour l'initialiser
user.getPortfolios().size();
// appeler la méthode « get » de la propriété persistante à charger
user.getCVContent();
```

Dans cet exemple, le chargement de la collection est lancé lors de l'appel à la méthode `size()` de celle-ci. De même, le chargement de la propriété `cvContent` s'effectue lors de l'accès à celle-ci.

> **Attention :** le chargement dynamique ne fonctionne que sur une entité *managée* !

Une autre solution consiste à utiliser les requêtes EJB-QL avec des jointures explicites pour récupérer les entités déjà initialisées. Pour cela, il faut utiliser le mot clé `FETCH JOIN` dans la requête (voir chapitre 7).

Précision sur le fonctionnement du « lazy loading »

Comment les fournisseurs de persistance peuvent-ils intervenir dynamiquement alors que la classe de l'Entity Bean est un POJO ?

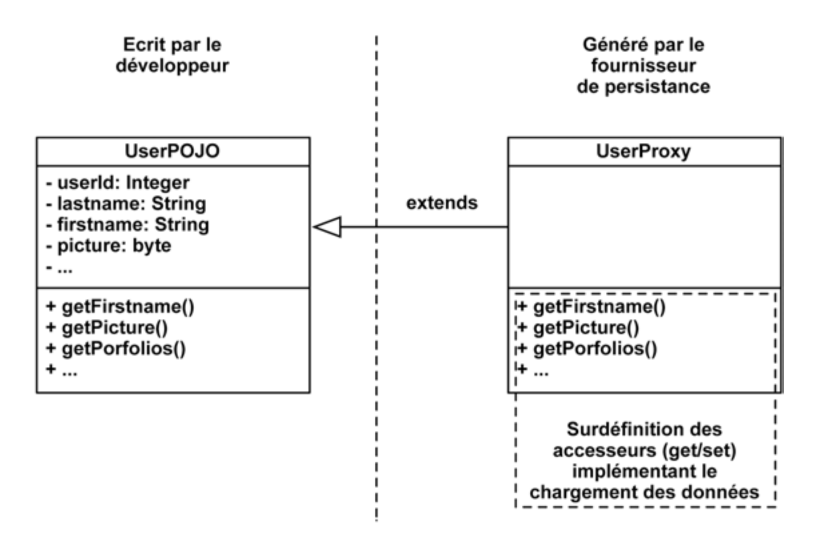


Figure 6.13 — Différentes stratégies d'insertion de code

Il existe trois grandes solutions (fig. 6.13) :

- Le système de crochet (*Hook*).
- L'ajout de *bytecode* post-compilation.
- L'utilisation d'un *proxy*.

Le principe reste cependant le même : du code est ajouté pour traiter et vérifier le *lazy-loading*. Par exemple, il est possible pour un fournisseur de persistance de surdéfinir les accesseurs dans une classe *proxy* (pour gérer les propriétés persistantes marquées comme chargement *lazy*). La gestion des collections est alors facilitée par l'utilisation d'une implémentation spécifique par le serveur d'applications (mais, bien entendu, non connu du développeur qui utilise les interfaces standard : `java.util.Collection`, `java.util.Set`...).

En résumé

Complexe, dans son approche, l'unité de persistance facilite énormément la vie du développeur, en lui offrant toutes les opérations nécessaires pour l'accès et le maniement des données. C'est un outil exceptionnel pour mettre en place très rapidement un système de persistance fiable et robuste, au sein d'une application Java EE ou Java SE.

7

L'EJB-QL, le SQL selon EJB

Objectif

Depuis déjà longtemps, SQL est le langage de référence pour exécuter des requêtes sur des bases de données relationnelles. Certes performant et standardisé, celui-ci ne correspond plus aux langages de développement orientés objets.

Les Mapping Objets/Relationnel ont alors vu le jour. Se dotant de dérivés de SQL permettant d'exécuter des requêtes sur des objets, ces systèmes sont très vite devenus référents en la matière. C'est le cas d'EJB-QL qui permet d'utiliser directement les Beans dans les requêtes.

7.1 INTRODUCTION

Nous présenterons de façon simple mais concise la manière d'utiliser l'EJB-QL dans la spécification EJB 2. Nous expliquerons ensuite, de manière détaillée, l'utilisation de l'EJB-QL dans un environnement compatible avec la spécification EJB 3.

7.1.1 Qu'est-ce que EJB-QL ?

EJB-QL (*Entreprise JavaBean Query Language*) est une spécification de langage de requêtes. Intégré au système EJB, ce langage de requêtes présente plusieurs avantages. Tout d'abord sa portabilité qui lui permet d'être utilisé à l'identique entre les différentes versions du langage SQL. En effet, celui-ci repose sur une abstraction du langage SQL et est traduit en « vrai » SQL lors de son exécution. D'autre part, l'EJB-

QL permet d'utiliser les objets des Entity Beans de l'application directement dans les requêtes, pour plus de facilité. De même, le fournisseur de persistance traduira ces requêtes en « vrai » SQL lors de l'exécution.

L'EJB-QL existe depuis la spécification EJB 2.0 et a subi quelques évolutions avec la version 2.1. Toutefois, celle-ci a souvent été critiquée pour son manque de fonctionnalités. La spécification 3.0 s'inspire fortement des langages de requête objet tels HSQL (Hibernate SQL) ou JDOQL.

La finalité de ce langage est en quelque sorte identique à celle du gestionnaire de persistance, qui offre, en effet, la possibilité d'effectuer des requêtes de type INSERT, UPDATE ou encore DELETE.

7.1.2 Le schéma abstrait

Afin de travailler avec les entités, EJB-QL s'appuie sur le schéma abstrait des Entity Beans. Le schéma abstrait est une représentation interne des entités et de leurs relations. Toutefois, même si ce terme semble complexe à première vue, on l'assimile couramment au nom de l'Entity Bean.

Ce nom est défini dans le descripteur de déploiement « ejb-jar.xml » par la balise `<abstract-schema-name>`.

```
<entity >
 <abstract-schema-name>AccountInfo</abstract-schema-name>
...
```

Avec la spécification EJB 3, on utilise généralement les annotations. On peut alors spécifier le nom de référence par l'attribut `name` de l'annotation `@Entity`.

```
@Entity(name = "AccountInfo")
public class AccountInfo { ... }
```

Par défaut, le nom de la classe de l'Entity Bean sera utilisé. Par exemple, le schéma abstrait par défaut d'un Entity Bean, défini par la classe `User`, est « User ».

Le terme abstrait distingue ce schéma du schéma physique de la base de données. Dans une base de données relationnelle, le schéma physique correspond à la structure des tables et colonnes. En EJB-QL vous travaillez avec les schémas abstraits et non avec le nom des tables de votre schéma physique, comme le représente la figure 7.1.

> **Conseil :** nous vous conseillons de dessiner un croquis du schéma abstrait avant d'écrire vos requêtes.

En résumé, il faut retenir qu'il s'agit d'une représentation abstraite des Entity Beans, utilisé par EJB-QL pour naviguer entre les propriétés persistantes et relationnelles.

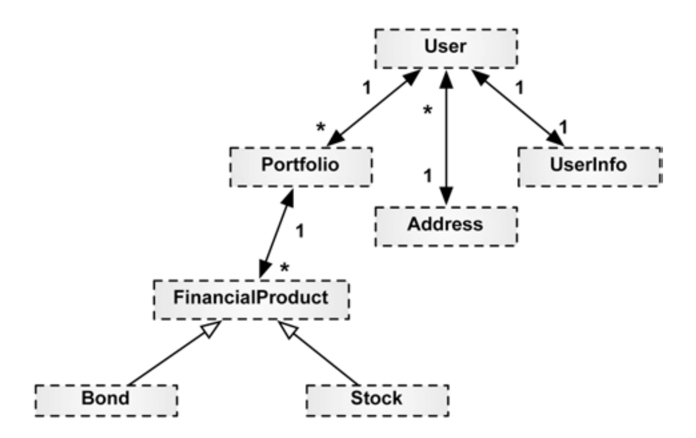


Figure 7.1 — Vue complète d'un schéma abstrait

7.2 EJB-QL POUR EJB 2

L'utilisation de requêtes de type EJB-QL n'a jamais été très simple en EJB 2. En effet, nous allons voir qu'il faut travailler avec les trois éléments des Entity Beans EJB 2 (interface métier, interface home, descripteur de déploiement). De plus, seules les requêtes de sélection/recherche sont disponibles dans cette spécification.

7.2.1 Le langage EJB-QL 2

L'EJB-QL a été standardisé à partir de la spécification 2.0. Nous ne rentrerons pas dans les détails d'implémentation de cette version car la version 3, qui est expliquée plus loin (paragraphe 7.3), reprend l'ensemble des concepts.

Nous détaillerons cependant, quelques exemples et principes de base de l'EJB-QL 2, et nous finirons par voir les limites de ce langage dans cette version.

Les requêtes simples

Une requête simple, est une requête ne possédant pas de clause `WHERE`. Elles sont généralement utilisées pour récupérer un ensemble complet d'instances. Voici quelques exemples :

```
SELECT OBJECT( u ) FROM User AS u
SELECT OBJECT( user ) FROM User AS user
SELECT us.firstname FROM User us
```

La clause `FROM` détermine quel Entity Bean est inclus dans la requête `SELECT`. Elle fournit l'étendue de la sélection. Dans notre exemple, la clause `FROM` déclare le type « User » qui représente le schéma abstrait de l'Entity Bean `User`.

La partie `AS us` permet d'assigner un identifiant à l'Entity Bean `User` au sein de la requête. L'instruction `AS` n'est pas obligatoire et peut être omise. Cela est similaire au SQL (association d'un identifiant à une table). Un identifiant n'a pas de taille limitée et suit les mêmes règles que pour les noms de variable en Java.

L'opérateur `OBJECT()` est requis lorsque la requête `SELECT` retourne une entité. Dans la troisième requête, cette instruction est omise car la requête retourne la propriété `firstname` de `User` et non l'entité `User` au complet.

Travailler avec les objets

L'EJB-QL permet à la clause `SELECT` de retourner n'importe quel Entity Bean ou un simple champ persistant ou relationnel. En voici un exemple :

```
SELECT u.firstname FROM User u
SELECT u.address.city FROM User u
```

La clause `SELECT` utilise un chemin « simple » pour sélectionner et retourner, dans l'exemple, le nom de famille d'un utilisateur. Il est possible d'utiliser les champs CMP (*Container Managed Persistence*) et CMR (*Container Managed Relationship*). Ces champs sont déclarés dans le descripteur de déploiement *via* les balises <cmp-field> et <cmr-field>.

La navigation se fait de la même façon qu'en langage Java, l'opérateur « . » (point) permet de spécifier une propriété à utiliser. La longueur du chemin dépend du nombre de niveau d'agrégation de vos objets.

Cependant la clause `SELECT` doit toujours se terminer par un type simple. C'est-à-dire, qu'il vous est interdit, par exemple, de sélectionner une propriété multi-valuées (une collection). Voici un exemple d'illégalité :

```
//interdit
SELECT u.portfolios FROM User u
SELECT u.portfolios.name FROM User u
```

Le champ relationnel `portfolios` étant du type `java.util.Collection`, il est interdit de le retourner *via* la requête `SELECT`, à l'image du langage Java qui interdit lui-même l'accès aux objets directement *via* une `Collection`.

```
// impossible
client.getReservations().getCroisiere()
```

L'astuce pour contourner ce problème est d'utiliser une jointure explicite ou d'utiliser l'instruction IN (expliquée en détail dans la partie EJB-QL 3).

Limites de EJB-QL 2

Ce langage a souvent été critiqué pour son manque de fonctionnalités. Voici les détails des différentes limites et reproches faites à ce langage :

- `OBJECT()` : cet opérateur ne semble pas réellement nécessaire. En effet, il serait facile pour un fournisseur d'EJB de déterminer lorsqu'un type d'« abstract schema » est retourné. Son utilisation n'est d'ailleurs pas très claire : il est

requis lorsqu'un type de retour est un Entity Bean mais ne l'est pas lorsqu'on spécifie un chemin, dans la clause `SELECT`, qui pointe vers un champ relationnel (CMR). Dans les deux cas, la requête retourne un Entity Bean.

- `ORDER BY` manquante : vous avez pu remarquer que nous n'avons pas parlé de cette clause bien connue en SQL. En effet, celle-ci n'existe pas dans l'EJB-QL 2, bien que l'ordonnancement des résultats soit très important dans n'importe quel langage de requête.
- Support des dates manquant : l'EJB-QL 2, ne supporte pas les champs de type `java.util.Date` ; ce qui est un inconvénient non négligeable.
  Pour utiliser les dates, vous devez utiliser le type « Long » et donc enregistrer les dates en secondes. Vous devrez alors effectuer un ensemble d'opérations manuelles pour travailler sur ces dates (comparaison, soustraction…).
- Fonctions d'expression limitées : l'EJB-QL 2 ne fournit qu'un ensemble très restreint de fonctions d'expression. Par exemple, la fonction `COUNT()` n'est pas implémentée.
- Un unique type de requête : seules les requêtes `SELECT` sont disponibles dans cette version.

Heureusement, la nouvelle version EJB-QL 3 comble en grande partie ces manques.

7.2.2 Utilisation en EJB 2

La spécification EJB 2 oblige le développeur à travailler avec des requêtes EJB-QL dites « statiques ». Cela signifie qu'elles sont définies lors du développement et ne peuvent être générées dynamiquement lors de l'exécution de l'application.

Pour cela, le développeur a la possibilité de déclarer deux types de méthode de recherche :

- Les méthodes de type « find », qui sont accessibles depuis l'interface « home ». Elles sont généralement préfixées avec « findBy ».
- Les méthodes de type « select », utilisées en interne dans l'Entity Bean, pour l'élaboration des tâches métiers.

Nous présentons, ici, les différentes méthodes, leur création et utilisation dans une application.

Les méthodes « find »

Il s'agit des méthodes pouvant être appelées par les clients des Entity Beans. On parle, de façon plus technique, de « finders » pour qualifier ces méthodes. Elles sont disponibles dans l'interface « home » et sont donc utilisées pour la récupération d'instances de l'Entity Bean.

Remarque : l'interface « home » étant l'interface de fabrication d'Entity Beans, il semble alors logique de l'utiliser pour récupérer des instances de celui-ci.

Prenons l'exemple de l'Entity Bean `User` défini par la classe `UserBean`. Nous souhaitons ajouter deux méthodes de récupération d'utilisateur :

- `findByRange(Integer idMin, Integer idMax)` afin de récupérer tout utilisateur ayant une clé primaire entre `idMin` et `idMax`.
- `findUserByLogin(String login)` afin de récupérer un Entity Bean ayant pour `login` celui spécifié par le paramètre

La première étape consiste à ajouter ces méthodes dans l'interface *home* (ici locale) :

```
public interface UserHomeLocal extends javax.ejb.EJBLocalHome {
//...
  public UserLocal findByPrimaryKey(Integer primaryKey)
    throws javax.ejb.FinderException;

  public java.util.Collection findByRange( Integer idMin, Integer idMax)
    throws javax.ejb.FinderException;

  public UserLocal findUserByLogin(String lastname)
    throws javax.ejb.FinderException ;
}
```

Lorsque la requête peut renvoyer plusieurs instances de l'entité, on utilise le type `java.util.Collection` ou `java.util.Set` suivant les exigences. Si aucun élément n'est trouvé, une `Collection/Set` vide est retournée.

Si vous souhaitez déclarer ces « finders » dans l'interface « home remote », il vous faut ajouter l'exception `java.rmi.RemoteException` à la clause `throws` afin de parer à tout problème réseau ou d'échange d'informations.

Attention : il est déconseillé d'utiliser l'accès « remote » pour un Entity Bean.

On peut remarquer la présence de la méthode `findByPrimaryKey()`, qui est systématique dans l'interface « home » d'un Entity Bean. Elle permet de retrouver une instance d'un Entity Bean à partir de son identifiant (clé primaire).

La seconde étape est d'ajouter une déclaration dans le descripteur de déploiement « ejb-jar.xml ». Voici la déclaration associée à l'exemple précédent.

```
...
<query>
  <query-method>
    <method-name>findByLogin</method-name>
      <method-params>
```

```
        <method-param>
          java.lang.String
        </method-param>
      </method-params>
  </query-method>
  <result-type-mapping>Local</result-type-mapping>
  <ejb-ql>
     SELECT OBJECT(user) FROM User AS user
     WHERE user.login = ?1;
   </ejb-ql>
</query>
<query>
  <query-method>
    <method-name>
      findByRange
    </method-name>
    <method-params>
      <method-param>
        java.langInteger
      </method-param>
      <method-param>
        java.langInteger
      </method-param>
    </method-params>
  </query-method>
  <result-type-mapping>Local</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(user) FROM User AS user
    WHERE user.userId > ?1 AND user.userId < ?2;
  </ejb-ql>
</query>
```

Pour chaque méthode de recherche, il faut ajouter un élément `<query>` (sous-balise de `<entity>`). Dans `<query-method>`, on spécifie le nom de la méthode ainsi que ses paramètres. La balise `<ejb-ql>` contient la requête EJB-QL associée à la méthode.

Les paramètres sont inclus dans la requête à la place des « `?x` » où `x` est un entier commençant par 1. L'ordre dans lequel les paramètres sont spécifiés est donc très important, car `?1` inclut le premier, `?2` le second, etc.

Remarque : l'utilisation de requêtes paramétrées n'est pas chose nouvelle. En effet, elles existent également avec JDBC *via* les `PreparedStatement`.

La balise `<result-type-mapping>` spécifie si la ou les instance(s) retournée(s) sont de type « remote » ou « local ». On travaille généralement avec des retours de type « local » car le type « remote » pose de nombreux problèmes, principalement au niveau des relations inter Entity Bean.

Les méthodes « select »

Celles-ci sont similaires aux méthodes de recherche, cependant, elles sont réservées aux usages internes des Entity Beans. Ainsi, on les utilise généralement dans les méthodes de « callbacks » (`ejbLoad()`, `ejbStore()...`) et dans les méthodes métiers.

Contrairement aux « finders », les méthodes « select » sont déclarées dans la classe de l'Entity Bean. Elles doivent respecter le préfixe « ejbSelect » afin d'être reconnues par le conteneur lors du déploiement.

Voici l'exemple d'une méthode « select » permettant de récupérer l'ensemble des `User` :

```
public class UserBean implements javax.ejb.EntityBean {
  public abstract java.util.Set ejbSelectAll()
    throws javax.ejb.FinderException;
}
```

La méthode `ejbSelectAll()` est abstraite et retourne donc un ensemble de type `java.util.Set`. L'implémentation de celle-ci est à la charge du conteneur lors du déploiement.

Il faut ensuite déclarer la méthode dans le descripteur de déploiement. Nous procédons alors comme pour les « finders ».

```
<query>
  <query-method>
    <method-name>ejbSelectAll</method-name>
    <method-params />
  <query-method>
  <ejb-ql>
    SELECT OBJECT(user) FROM User AS user
  </ejb-ql>
</query>
```

Ces méthodes renvoient des instances locales ou distantes de l'Entity Bean, en fonction du contexte dans lequel la méthode est appelée.

7.3 EJB-QL POUR EJB 3

Comme vu précédemment, le langage EJB-QL, dans sa version 2, est très limité, et cette nouvelle version peut être considérée comme une vraie évolution. En effet, ce langage se base sur les existants qui ont été développés par la communauté en parallèle des spécifications. Hibernate, et son HSQL, a sans doute été une base solide pour l'élaboration de celle-ci.

Dans cette version, il est désormais possible d'utiliser de nombreuses fonctionnalités supplémentaires : GROUP BY, HAVING, sous-requêtes… L'une des grandes nouveautés est sans doute la possibilité de créer des requêtes à la volée (ou « dynamiques ») au moment de l'exécution.

Avant de rentrer dans les détails innombrables de ce langage, voici un premier exemple simple qui vous permettra de comprendre plus facilement la suite des explications.

```
String queryString = "SELECT user FROM User AS user";
Query query = entityManager.createQuery(queryString);
```

```
List<User> allUsers = query.resultList();

for (User u : allUsers ) {
  // Traitements sur l'utilisateur …
}
```

L'exemple précédent utilise trois concepts différents : le langage EJB-QL, l'Entity Manager et l'API Query.

Celui-ci retourne l'ensemble des instances de l'Entity Bean `User` dans une liste. Celle-ci peut alors être parcourue par un `foreach`. Les détails d'utilisation de ces éléments sont expliqués tout au long de cette partie.

Suivant les vendeurs, il est possible de journaliser (« logs ») les requêtes SQL exécutées par le gestionnaire de persistance. Par exemple, il suffit d'ajouter la propriété « hibernate.show_sql » avec la valeur « true » si vous utilisez le moteur de persistance Hibernate. Nous essayerons au maximum d'illustrer les requêtes d'exemple avec leur correspondance SQL.

Attention : la correspondance SQL a été réalisée à partir de tests effectués avec le fournisseur Hibernate et une base de données Mysql.

7.3.1 Le langage EJB-QL 3

Une requête EJB-QL peut remplir une tâche de sélection (`SELECT`), modification (`UPDATE`) ou suppression (`DELETE`).

Une requête EJB-QL est similaire à du SQL, avec les clauses suivantes :

- `SELECT` : liste les Entity Beans et les propriétés retournées par la requête.
- `FROM` : définit les Entity Beans utilisés. Ceux-ci doivent être déclarés *via* l'expression `AS`.
- `WHERE` : permet d'appliquer des critères de recherche. Il est possible d'y spécifier aussi bien des types Java ayant leur équivalent dans les bases de données (String, Integer, Double…) mais également des Entity Beans. Dans ce dernier cas, lors du passage en requête native, le critère s'appliquera sur la clé primaire.

Bien sûr, d'autres pourront être rajoutées, comme nous allons le voir, mais celles-ci sont quasi systématiquement utilisées pour récupérer des données.

L'opérateur « . » sert à naviguer entre les propriétés et les relations des Entity Beans. Par exemple, afin de récupérer un portefeuille dont l'utilisateur a pour id 5 :

```
SELECT portfolio FROM Portfolio AS portfolio WHERE portfolio.user.id=5
```

```
Correspondance en SQL : select portfolio0_.id as id38_, portfolio0_.name as
name38_, portfolio0.user_fk as user3_38_ from Portfolio portfolio0_ where
portfolio0_.user_fk=?
```

On devine assez clairement que l'utilisation du « . » est similaire, dans cet exemple, à appeler la méthode `Portfolio.getUser()`.

> **Attention** : le caractère de terminaison « ; » des requêtes SQL ne doit pas être utilisé, sous peine qu'une `PersistanceException` soit lancée.

Les requêtes SELECT

La requête la plus utilisée est sans doute SELECT, la requête de recherche. L'instruction à utiliser est `SELECT` comme l'illustre l'exemple suivant, où l'on sélectionne tous les utilisateurs :

```
SELECT usr FROM User AS usr
```

```
Correspondance en SQL : select user0_.id as id21_,
  user0_.address_fk as address8_21_, user0_.password as password21_,
  user0_.lastName as lastName21_, user0_.firstName as firstName21_,
  user0_.login as login21_, user0_.sex as sex21_,
  user0_.birthDate as birthDate21_ from XUser user0_
```

Il existe de nombreuses similitudes avec le langage SQL. Toutefois, il faut bien comprendre que la différence vient du fait qu'EJB-QL est orienté objet. Dans l'exemple précédent, « User » correspond au schéma abstrait de l'Entity Bean `User` et « usr » correspond à un alias (comme en SQL).

L'exemple précédent travaille directement avec l'objet `User`, cependant il est parfois pratique de ne récupérer que l'id, ou le nom, ou le prénom… Avec EJB 3, il est possible de spécifier les propriétés que l'on souhaite récupérer. Il suffit de les spécifier dans la clause `SELECT`.

```
SELECT user.id, user.lastName FROM User AS user
```

```
Correspondance en SQL : select user0_.id as col_0_0_,
  user0_.firstName as col_1_0 from XUser user0_
```

Les requêtes DELETE et UPDATE

Ces requêtes vont fournir un ensemble d'opérations pour modifier ou supprimer un Entity Bean. Leur particularité commune est de n'avoir qu'une seule entité dans la clause `FROM`. Ainsi, si vous souhaitez supprimer ou modifier des enregistrements de plusieurs Entity Beans différents, vous devez effectuer plusieurs requêtes. Voici deux exemples montrant l'utilisation de ces requêtes.

```
DELETE FROM User AS user
WHERE user.id = 5

Correspondance en SQL : delete from XUser where id=?
```

```
UPDATE User AS user
SET user.firstName = "Durand"
WHERE user.id = 1
```

```
Correspondance en SQL : update XUser set firstName=? where id=?
```

La clause `DELETE` d'EJB-QL ne supporte pas la suppression en cascade. En effet, même si la relation est configurée avec `CascadeType.REMOVE` ou `CascadeType.ALL`, il faudra tout de même écrire manuellement leur retrait de la base de données. Dans cette requête `DELETE`, nous risquons de recevoir des exceptions dans le cas où l'Entity Bean `User` posséderait des relations. Nous avons alors trois possibilités :

- Utiliser le gestionnaire de persistance qui applique la suppression en cascade.
- Écrire explicitement toutes les requêtes EJB-QL de suppressions dans le bon ordre.
- Utiliser les possibilités de la base de données et gérer la suppression en cascade au niveau de celle-ci.

Dans ce dernier cas, Il est possible d'utiliser l'expression ON DELETE CASCADE pour, par exemple, les bases de données de type MySQL version 5 et du moteur de stockage InnoDB.

Il vous faudra, auparavant, vérifier que les contraintes de modifications en cascade ont bien été appliquées. Voici un extrait du code SQL à utiliser :

```
-- Contraintes pour la table `AccountInfo`
ALTER TABLE `AccountInfo`
  ADD CONSTRAINT `fk1` FOREIGN KEY (`id`)
  REFERENCES `xuser` (`id`) ON DELETE CASCADE;

-- Contraintes pour la table `FINANCIALPRODUCT`
ALTER TABLE `FINANCIALPRODUCT`
  ADD CONSTRAINT `fk2` FOREIGN KEY (`portfolio_fk`)
  REFERENCES `portfolio` (`id`) ON DELETE CASCADE;

-- Contraintes pour la table `Portfolio`
ALTER TABLE `Portfolio`
  ADD CONSTRAINT `fk3` FOREIGN KEY (`user_fk`)
  REFERENCES `xuser` (`id`) ON DELETE CASCADE;
```

De cette façon, lors de la suppression d'un utilisateur, tous ses portefeuilles, produits financiers, et informations seront automatiquement supprimés.

La clause `WHERE`

Les permissions incluses dans la clause `WHERE` existent pour la plupart dans une forme similaire à celle que l'on peut utiliser en SQL. Voici une liste résumant celles les plus couramment utilisées :

- BETWEEN : opérateur conditionnel permettant de restreindre les résultats suivant un intervalle. L'exemple suivant sélectionne tous les utilisateurs ayant un identifiant de 1500 à 2000.

```
SELECT user FROM User AS user
WHERE user.id BETWEEN 1500 AND 2000
```

```
Correspondance en SQL : select user0_.id as id4_,
  user0_.address_fk as address8_4_, user0_.password as password4_,
  user0_.lastName as lastName4_, user0_.firstName as firstName4_,
  user0_.login as login4_, user0_.sex as sex4_,
  user0_.birthDate as birthDate4_ from XUser user0_
  where user0_.id between ? and ?
```

- LIKE : permet de comparer la valeur d'un champ avec un motif spécifié. La requête suivante récupère, par exemple, tous les comptes ayant un prénom commençant par « jean ».

```
SELECT user FROM User AS user
WHERE user.firstName LIKE 'jean%'
```

```
Correspondance en SQL : select user0_.id as id4_,
  user0_.address_fk as address8_4_, user0_.password as password4_,
  user0_.lastName as lastName4_, user0_.firstName as firstName4_,
  user0_.login as login4_, user0_.sex as sex4_,
  user0_.birthDate as birthDate4_ from XUser user0_
  where user0_.firstName like ?
```

Le motif se construit avec deux caractères spéciaux. Le premier est « % » et il s'utilise pour un nombre de caractères indéfinis. Le second, « _ », représente un seul caractère.

Au cas où votre chaîne de caractère utiliserait réellement ces deux caractères, vous avez à votre disposition le caractère d'échappement « \ ». Voici quelques exemples d'utilisation :

```
firstname LIKE '_axime'
firstname LIKE 'Ma%'
misc LIKE '%\_%'
```

- La première ligne de cet exemple renverra les entrées telles que « Maxime », « Aaxime »... La seconde « Marc », « Martine », « Maxime »... et la dernière les entrées contenant le caractère « _ », comme « java_bean », « java_ », « _java »...
- IN : teste une appartenance à une liste de chaînes de caractères. La requête suivante récupère, par exemple, tous les contacts situés en France, Espagne, et Belgique.

```
SELECT user FROM User AS user
WHERE user.address.country IN  ('France', 'Spain', 'Belgium')
```

```
Correspondance en SQL : select user0_.id as id4_, user0_.address_fk as
address8_4_, user0_.password as password4_, user0_.lastName as lastName4_,
user0_.firstName as firstName4_, user0_.login as login4_, user0_.sex as sex4_,
user0_.birthDate as birthDate4_ from XUser user0_, Address address1_ where
user0_.address_fk=address1_.id and (address1_.country in ('France' , 'Espagne' ,
'Belgique'))
```

- `IS NULL` : teste si une valeur est nulle. Il s'agit de la valeur par défaut définie quand un champ n'a pas été renseigné. La requête suivante renvoie, par exemple, les enregistrements où le login n'a pas été spécifié.

```
SELECT user FROM User AS user
WHERE user.login IS NULL
```

```
Correspondance en SQL : select user0_.id as id21_,
  user0_.address_fk as address8_21_, user0_.password as password21_,
  user0_.lastName as lastName21_, user0_.firstName as firstName21_,
  user0_.login as login21_, user0_.sex as sex21_,
  user0_.birthDate as birthDate21_from XUser user0_where user0_.login is null
```

- `MEMBER OF` : teste l'appartenance d'une instance à une collection, tout comme la méthode `contains()` de l'interface `java.util.Collection`. L'exemple suivant récupère les utilisateurs possédant le portefeuille passé en paramètre (nous verrons le passage des paramètres plus loin dans ce chapitre).

```
SELECT user
FROM User AS user
WHERE ?1 MEMBER OF user.portfolios
```

```
Correspondance en SQL : select user0_.id as id21_,
  user0_.address_fk as address8_21_, user0_.password as password21_,
  user0_.lastName as lastName21_, user0_.firstName as firstName21_,
  user0_.login as login21_, user0_.sex as sex21_,
  user0_.birthDate as birthDate21_ from XUser user0_
  where ? in (select portfolios1_.id from Portfolio portfolios1_
  where user0_.id=portfolios1_.user_fk)
```

- `EMPTY` : teste si une collection est vide. Par exemple, nous allons ici retourner la liste des utilisateurs n'ayant aucun portefeuille d'action.

```
SELECT user
FROM User AS user
WHERE user.portfolios IS EMPTY
```

```
Correspondance en SQL : select user0_.id as id21_,
  user0_.address_fk as address8_21_, user0_.password as password21_,
  user0_.lastName as lastName21_, user0_.firstName as firstName21_,
  user0_.login as login21_, user0_.sex as sex21_,
  user0_.birthDate as birthDate21_ from XUser user0_
  where  not (exists (select portfolios1_.id
  from Portfolio portfolios1_ where user0_.id=portfolios1_.user_fk))
```

- `NOT` : inverse le résultat de la condition. Nous pouvons l'utiliser avec les précédents opérateurs (`NOT BETWEEN`, `NOT LIKE`, `NOT IN`, `IS NOT NULL`...).

La clause ORDER BY

La clause `ORDER BY` permet de ranger par ordre les résultats d'une requête, à partir d'un ou plusieurs champs en utilisant l'ordre alphanumérique puis alphabétique. Pour connaître exactement l'implémentation de cette fonction, il faut se référer au manuel de la base de données.

L'exemple suivant retourne la liste des utilisateurs ordonnés suivant leur identifiant.

```
SELECT user
FROM User AS user
ORDER BY user.id ASC
```

```
Correspondance en SQL : select user0_.id as id4_,
  user0_.address_fk as address8_4_, user0_.password as password4_,
  user0_.lastName as lastName4_, user0_.firstName as firstName4_,
  user0_.login as login4_, user0_.sex as sex4_,
  user0_.birthDate as birthDate4_ from XUser user0_ order by user0_.id
```

Le tableau 7.1 présente ce que l'on pourrait récupérer (suivant les données de la base de données).

Tableau 7.1 — Résultats d'un `ORDER BY`

Id	user.firstName
4	Cyril
6	Frédéric
7	Olivier
8	Maxime
10	Jean-Baptiste

Le mot-clé optionnel `ASC` signifie que le classement se fait de façon ascendante. Pour l'effectuer de façon descendante, il faut utiliser `DESC` (du plus grand au plus petit). Par défaut, c'est `ASC` qui est appliqué.

Il est bien entendu possible de combiner ces critères. Ainsi, la requête suivante affiche le nom par ordre croissant, cependant lorsque deux utilisateurs ont le même nom, le tri se fait ensuite sur le prénom, de façon croissante, puis décroissante dans le second exemple.

```
SELECT user
FROM User AS user
ORDER BY user.lastName, user.firstName ASC
```

```
Correspondance en SQL : select user0_.id as col_0_0_,
  user0_.firstName as col_1_0_ from XUser user0_
  order by user0_.lastName, user0_.firstName ASC
```

```
SELECT user
FROM User AS user
ORDER BY user.lastName ASC, user.firstName DESC
```

```
Correspondance en SQL : select user0_.id as col_0_0_,
  user0_.firstName as col_1_0_ from XUser user0_
  order by user0_.lastName ASC, user0_.firstName DESC
```

Les fonctions d'agrégation

Les fonctions d'agrégation servent à effectuer des opérations sur des ensembles. On les retrouve en EJB-QL *via* les instructions suivantes :

- `avg()` : retourne une moyenne par groupe.
- `count()` : retourne le nombre d'enregistrements.
- `max()` : retourne la valeur la plus élevée.
- `min()` : retourne la valeur la plus basse.
- `sum()` : retourne la somme des valeurs.

Par exemple, vous pouvez utiliser la fonction `count()` pour connaître le nombre d'utilisateurs inscrits dans votre base de données.

```
SELECT COUNT(user)
FROM User AS user
```

```
Correspondance en SQL : select count(user0_.id) as col_0_0_ from XUser user0_
```

Il est également possible d'utiliser la clause `GROUP BY` pour appliquer la fonction d'agrégation sur un lot d'enregistrements. Voici un exemple permettant de retourner le nombre de portefeuilles par utilisateur :

```
SELECT user.firstName, COUNT(user.portfolios) AS nbrPortfolio
FROM User AS user
```

```
GROUP BY user
```

```
Correspondance en SQL : select user0_.id as col_0_0_,
  count(portfolios1_.id) as col_1_0_, user0_.id as id72_,
  user0_.address_fk as address8_72_, user0_.password as password72_,
  user0_.lastName as lastName72_, user0_.firstName as firstName72_,
  user0_.login as login72_, user0_.sex as sex72_,
  user0_.birthDate as birthDate72_ from XUser user0_
  left outer join Portfolio portfolios1_
  on user0_.id=portfolios1_.user_fk group by user0_.id
```

Les résultats pourraient être les suivants, en supposant des données fictives (tableau 7.2):

Tableau 7.2 — Résultats d'un `COUNT(user.portfolios)`

user.firstName	nbrPortfolio
Cyril	2
Frédéric	4
Jean-Baptiste	5
Maxime	1
Olivier	9

L'intérêt du `GROUP BY` dans cette requête est de pouvoir appliquer la fonction `COUNT()` sur chaque utilisateur.

La clause `HAVING` permet de spécifier des critères, comme avec la clause `WHERE`, sur une colonne générée par une des fonctions d'agrégation. Dans l'exemple suivant, nous retournons seulement les informations lorsque le nombre de portefeuilles financiers est supérieur à 2.

```
SELECT user.firstName, COUNT(user.portfolios) AS nbrPortfolio
FROM User AS user
GROUP BY user.firstName
HAVING nbrPortfolio > 2
```

> **Remarque :** conformément aux normes SQL, il faut utiliser la clause `HAVING` afin d'appliquer des critères sur le résultat d'une fonction.

Manipuler les collections avec l'opérateur IN

La majeure partie des relations utilise des collections. Le parcours de leurs entrées est particulièrement intéressant car il n'existe pas dans la logique relationnelle du SQL.

Nous avons pour cela l'opérateur IN. Il doit être placé dans la clause FROM, et permet de déclarer un alias pour les entrées d'une collection. Par exemple, nous pouvons récupérer l'ensemble des portefeuilles créés, comme suit :

```
SELECT portfolio
FROM User AS user, IN (user.portfolios) portfolio
```

```
Correspondance en SQL : select portfolio0_.id as id139_,
  portfolio0_.name as name139_, portfolio0_.user_fk as user3_139_
  from Portfolio portfolio0_ where portfolio0_.user_fk=?
```

Nous regroupons les éléments contenus dans la collection user.portfolios, dans l'alias « portfolio ». De ce fait, la requête regroupe les portefeuilles de tous les utilisateurs et les charge.

Les identifiants dans la clause FROM sont déclarés de gauche à droite. Lorsqu'un identifiant est déclaré, on peut l'utiliser dans les déclarations suivantes. La requête suivante récupère tous les produits financiers possédés par l'utilisateur ayant l'identifiant 42.

```
SELECT fp
FROM User AS user,
  IN (user.portfolios) portfolio,
  IN (portfolio.financialProducts) fp
WHERE user.id = 42 AND fp.quantity > 10
```

```
Correspondance en SQL : select financialp2_.id as id104_,
  financialp2_.quantity as quantity104_,
  financialp2_.buyValue as buyValue104_,
  financialp2_.productName as productN5_104_,
  financialp2_.buyDate as buyDate104_,
  financialp2_.portfolio_fk as portfolio10_104_,
  financialp2_.monthDuration as monthDur7_104_,
  financialp2_.rate as rate104_, financialp2_.price as price104_,
  financialp2_.financialproduct_type as financial_104_
  from XUser user0_ inner join Portfolio portfolios1_
  on user0_.id=portfolios1_.user_fk inner join FINANCIALPRODUCT financialp2_
  on portfolios1_.id=financialp2_.portfolio_fk where user0_.id=?
```

Cet opérateur permet de sélectionner directement les éléments d'une relation et de travailler avec. Dans l'exemple précédent, la requête retourne les FinancialProducts ayant une quantité supérieure à 10 de l'utilisateur d'id 42 (tout portefeuille confondu).

Une seconde fonction, ELEMENTS, remplit le même rôle, mais se situe dans la clause SELECT. Voici une requête avec un résultat similaire au premier exemple.

```
SELECT ELEMENTS(user.portfolios)
FROM User AS user
```

```
Correspondance en SQL : (1) select portfolios1_.id as col_0_0_
from XUser user0_, Portfolio portfolios1_

(2) select portfolio0_.id as id139_3_, portfolio0_.name as name139_3_,
  portfolio0_.user_fk as user3_139_3_, user1_.id as id140_0_,
  user1_.address_fk as address8_140_0_, user1_.password as password140_0_,
  user1_.lastName as lastName140_0_, user1_.firstName as firstName140_0_,
  user1_.login as login140_0_, user1_.sex as sex140_0_,
  user1_.birthDate as birthDate140_0_, address2_.id as id143_1_,
  address2_.country as country143_1_, address2_.city as city143_1_,
  address2_.numero as numero143_1_, address2_.street as street143_1_,
  address2_.zipCode as zipCode143_1_, accountinf3_.id as id142_2_,
  accountinf3_.accountId as accountId142_2_,
  accountinf3_.cardNumber as cardNumber142_2_,
  accountinf3_.amount as amount142_2_ from Portfolio portfolio0_
  left outer join XUser user1_ on portfolio0_.user_fk=user1_.id
  left outer join Address address2_ on user1_.address_fk=address2_.id
  left outer join AccountInfo accountinf3_ on user1_.id=accountinf3_.id
  where portfolio0_.id=?
```

Les jointures

Les jointures permettent de manipuler les relations entre les entités. Nous verrons que les fonctionnalités disponibles sont multiples et pourront s'adapter à de nombreux besoins. Pour illustrer cette déclaration, nous présentons la récupération des portefeuilles d'actions, possédés par un utilisateur, de différentes manières.

- En utilisant le style « thêta », avec la relation entre clés primaires et clés étrangères :

```
SELECT user.firstName, ai.cardNumber
FROM User AS user, AccountInfo AS ai
WHERE u.accountInfo.id = ai.id
```

```
Correspondance en SQL : select user0_.firstName as col_0_0_,
  accountinf1_.cardNumber as col_2_0_ from XUser user0_, AccountInfo
  accountinf1_ where user0_.id=accountinf1_.id
```

- En utilisant les propriétés relationnelles :

```
SELECT user.firstName, user.accountInfo.cardNumber
FROM User AS user
```

```
Correspondance en SQL : select user0_.firstName as col_0_0_,
  accountinf1_.cardNumber as col_2_0_ from XUser user0_, AccountInfo
  accountinf1_ where user0_.id=accountinf1_.id
```

- En utilisant une jointure *via* l'instruction `JOIN` :

```
SELECT user.firstName, ai.cardNumber
FROM User AS user
JOIN user.accountInfo AS ai
```

```
Correspondance en SQL : select user0_.firstName as col_0_0_,
  accountinf1_.cardNumber as col_2_0_ from XUser user0_ inner join AccountInfo
  accountinf1_ on user0_.id=accountinf1_.id
```

Tableau 7.3 — Résultats d'un `JOIN`

user.firstName	cardNumber
Cyril	1111-2222-3333-4444
Ophélie	2222-3333-4444-5555

Les résultats retournés sont les mêmes pour chacune des requêtes à quelques exceptions près. En effet, le fonctionnement des jointures est subtil. Lorsqu'une entité `User` ne possède pas de relation avec l'entité `AccountInfo`, l'enregistrement ne sera pas retourné !

LEFT JOIN

Afin de renvoyer les enregistrements de l'entité `Account`, même s'ils n'ont pas de relation, il faut effectuer une jointure dite « ouverte ». Pour cela, on utilise les instructions `LEFT OUTER JOIN` ou `RIGHT OUTER JOIN`, également existantes en SQL.

En EJB-QL, l'expression `LEFT JOIN` permet d'inclure systématiquement la première entité dans le résultat de la requête. `RIGHT JOIN`, quant à elle, ajoute les résultats de la seconde entité.

Remarque : l'instruction OUTER, utilisée en SQL, est devenue optionnelle.

L'exemple suivant utilise une jointure ouverte. Les résultats sont alors différents.

```
SELECT user.firstName, user.lastName, ai.cardNumber
FROM User AS user
LEFT JOIN user.accountInfo AS ai
```

Tableau 7.4 — Résultats d'un `LEFT JOIN`

user.firstName	cardNumber
Cyril	*1111-2222-3333-4444*
Maxime	*2222-3333-4444-5555*
Jean-Baptiste	*NULL*

En effet, la requête a désormais pris en compte le troisième enregistrement, qui n'a pas de relation avec `AccountInfo`. Cette requête exécute une « jointure ouverte sur la gauche ». Voici sa correspondance en SQL :

```
select user0_.firstName as col_0_0_, user0_.lastName as col_1_0_,
  accountinf1_.cardNumber as col_2_0_
  from User user0_
    left outer join AccountInfo accountinf1_ on
    user0_.accountinfo_fk=accountinf1_.id
```

INNER JOIN

Nous avons vu précédemment l'utilisation de l'opérateur `IN`. La jointure de type « INNER » provoque le même résultat. Elle est cependant plus familière aux développeurs utilisant couramment le SQL.

Il suffit d'utiliser le mot clé `INNER JOIN` pour réaliser ce type de jointure.

```
SELECT p FROM User u INNER JOIN u.portfolios p
```

```
Correspondance en SQL : select user0_.firstName as col_0_0_,
  user0_.lastName as col_1_0_, accountinf1_.cardNumber as col_2_0_
  from XUser user0_ inner join AccountInfo accountinf1_
  on user0_.id=accountinf1_.id
```

La requête précédente retourne l'ensemble des portefeuilles créés quel que soit l'utilisateur. Il est toutefois plus simple d'utiliser l'opérateur `IN`. Voici la correspondance utilisant « IN » :

```
SELECT p FROM User u, IN(u.portfolios) p
```

FETCH JOIN

Nous avons vu précédemment qu'il était possible d'utiliser un mécanisme du *lazy loading* (voir paragraphe 6.6.4). Dans ce cas, lorsque l'application accède à une propriété marquée avec *lazy*, le fournisseur de persistance doit la charger.

```
Query query = entityManager.createQuery("SELECT u FROM User u"

  +"WHERE u.id = :userId") ;
User user = (User) query.setParameter("userId", 1).getSingleResult();
Iterator<Portfolio> portfoliosIt = user.getPortfolios().iterator();
while(portfoliosIt.hasNext()) {
  Portfolio p = portfoliosIt.next();
  p.getFinancialProducts().size();
}
```

Cet exemple charge l'utilisateur d'id 1 et initialise ses portefeuilles et les produits financiers associés. Cela pose cependant des problèmes de performances. En effet, le moteur de persistance doit exécuter une requête à la ligne 2 puis pour chaque chargement des produits financiers associés (ligne 5). Celui-ci opère alors N+1 requêtes (N étant le nombre de portefeuilles).

Pour optimiser cela, il faut utiliser une requête EJB-QL avec une jointure de type « FETCH ».

```
SELECT u FROM User u INNER JOIN FETCH u.portfolio p LEFT JOIN FETCH
p.financalProducts
```

```
Correspondance en SQL : select user0_.id as id310_0_,
  portfolios1_.id as id309_1_, user0_.address_fk as address8_310_0_,
  user0_.password as password310_0_, user0_.lastName as lastName310_0_,
  user0_.firstName as firstName310_0_, user0_.login as login310_0_,
  user0_.sex as sex310_0_, user0_.birthDate as birthDate310_0_,
  portfolios1_.name as name309_1_, portfolios1_.user_fk as user3_309_1_,
  portfolios1_.user_fk as user3_0__, portfolios1_.id as id0__
  from XUser user0_ left outer join Portfolio portfolios1_
  on user0_.id=portfolios1_.user_fk
  left outer join FINANCIALPRODUCT financialp2_
  on portfolios1_.id=financialp2_.portfolio_fk
```

Une jointure « fetch » n'a normalement pas besoin de définir d'alias sauf lorsque vous souhaitez utiliser le concept du « fetch » récursivement.

Grâce à cette modification, une seule requête est exécutée. L'utilisation de ce type de jointure dans un programme est une nécessité pour l'optimisation et l'amélioration des performances. En effet, il est fortement conseillé d'éviter trop d'allers et retours avec la base de données.

Gestion du polymorphisme

EJB-QL supporte nativement le polymorphisme, c'est-à-dire les requêtes portant sur une hiérarchie d'objets.

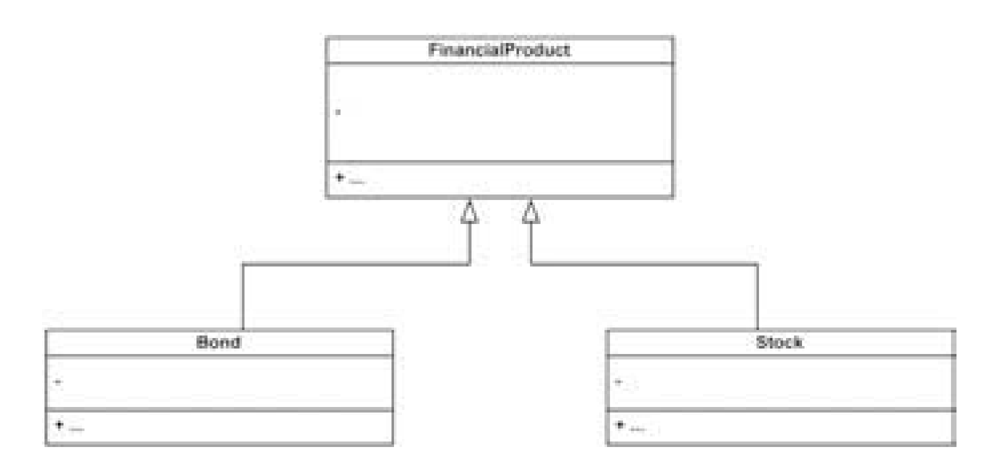


Figure 7.2 — Hiérarchie utilisée dans les exemples suivants

L'exécution de la requête suivante retourne tous les enregistrements liés à l'entité `FinancialProduct` et donc de ses sous-classes (fig.7.2).

```
SELECT fi FROM FinancialProduct AS fi
```

La collection de résultat contient des objets de type `Bond` ou `Stock`, c'est-à-dire les types concrets de la hiérarchie.

Sous-requêtes

Les sous-requêtes peuvent êtres placées dans les clauses `WHERE` et `HAVING`. Comme leur équivalent SQL, elles permettent d'imbriquer les requêtes. L'exemple suivant sélectionne les utilisateurs ayant plus de trois portefeuilles financiers.

```
SELECT user
FROM User user
WHERE (
  SELECT COUNT(portfolio)
  FROM Portfolio AS portfolio
  WHERE portfolio.user=user
  GROUP BY user
) >3
```

```
Correspondance en SQL : select user0_.id as id327_,
  user0_.address_fk as address8_327_, user0_.password as password327_,
  user0_.lastName as lastName327_, user0_.firstName as firstName327_,
  user0_.login as login327_, user0_.sex as sex327_,
  user0_.birthDate as birthDate327_ from XUser user0_
  where (select count(portfolio1_.id) from Portfolio portfolio1_
  where portfolio1_.user_fk=user0_.id group by user0_.id)>?
```

On remarque que l'on réutilise l'alias `user` de la requête principale dans la sous-requête. Un des principaux intérêts des sous-requêtes est de pouvoir se substituer aux jointures, notamment dans les requêtes de type `UPDATE` et `DELETE`.

La requête suivante supprime tous les produits financiers où l'utilisateur a un id ayant pour valeur 5 :

```
DELETE FROM FinancialProduct AS fp
WHERE fp.portfolio IN (
  SELECT portfolio
  FROM Portfolio AS portfolio
  WHERE portfolio.user IN (
    SELECT user
    FROM User AS user
    WHERE user.login = 5
  )
)
```

```
Correspondance en SQL : delete from FINANCIALPRODUCT
  where portfolio_fk in (
    select id from Portfolio portfolio1_ where portfolio1_.user_fk in (
      select id from XUser user2_ where user2_.id = 5
    )
)
```

Conseil : utilisez les sous-requêtes lorsque vous ne pouvez pas faire autrement. Nous vous conseillons d'étudier, d'abord, l'utilisation de jointures ou d'instructions telles que `IN`, `ELEMENTS`… qui sont d'ailleurs obligatoires dans des systèmes ne prenant pas en compte les sous-requêtes.

Lorsqu'une sous-requête est susceptible de retourner plusieurs lignes, il est alors possible de quantifier le résultat devant être retourné. Pour cela, on utilise les expressions :

- ALL retourne *vraie* si tous les résultats de la requête vérifient la condition.
- SOME retourne *vraie* si au moins un résultat vérifie la condition.
- ANY retourne *vraie* si aucun résultat ne vérifie la condition.

7.3.2 L'API Query

L'API Query est le point essentiel pour la jonction entre l'application et l'exécution de requêtes EJB-QL. Les méthodes de celle-ci sont regroupées dans l'interface javax.persistence.Query.

L'intérêt principal de cette API est de pouvoir créer des requêtes dynamiques sous forme de simples chaînes de caractères, et non plus de manière statique dans le descripteur de déploiement.

Il existe différents moyens de récupérer un objet de type Query. Les méthodes correspondantes sont regroupées dans l'interface EntityManager (chapitre 6). Voici un exemple simple :

```
Query query = entityManager.createQuery("SELECT user FROM User As user");
List<User> listUsers = query.getResultList();
```

Nous récupérons ici un objet Query *via* la méthode createQuery() de l'EntityManager. Il suffit ensuite d'appeler la méthode getResultList() pour exécuter la requête et récupérer le résultat de celle-ci.

Voici une description des différentes méthodes classiques de l'interface Query.

java.util.List getResultList()

Cette méthode exécute la requête et retourne l'ensemble des résultats de celle-ci. Elle permet par exemple de récupérer un ensemble d'Entity Beans complet. Il est alors possible de spécifier plusieurs propriétés dans la clause SELECT afin de récupérer seulement certaines valeurs de l'entité, sous forme de tableau.

```
Query query =
entityManager.createQuery("SELECT user.id, user.lastName FROM User As user");
List<Object[]> listUsers = query.getResultList();
for(Object[] valueArray : listUsers){
  Integer id = (Integer) valueArray[0];
  String name = (String) valueArray[1];
}
```

Dans ce cas-ci, la méthode getResultList() retourne une liste de tableau d'objet (« Object[] »).

java.lang.Object getSingleResult()

Cette méthode exécute la requête et retourne un unique résultat. Si le nombre de résultats est supérieur à 1, une exception `NonUniqueResultException` est levée. À l'inverse, si aucun résultat n'est trouvé, une exception de type `EntityNotFoundException` est levée.

Cette méthode doit être utilisée seulement si vous êtes sûr que la requête ne retourne qu'un unique résultat.

```
Query query = entityManager.createQuery("SELECT user FROM User AS user WHERE

  user.login = 'durand.dupont'");
User currentUser = (User) query.getSingleResult();
```

Dans cet exemple, la propriété `login` est unique. De ce fait, nous sommes sûrs que la méthode ne peut renvoyer plusieurs résultats.

Query setMaxResults(int max) et Query setFirstResult(int first)

Ces deux méthodes définissent respectivement le nombre maximal de résultats et l'index du premier élément à retourner.

Celles-ci permettent de limiter les résultats de la requête. En effet, si la requête tente de retourner plusieurs milliers de lignes, vous risquez de réduire rapidement les performances de votre application. De plus, on ne travaille généralement pas avec l'ensemble des résultats d'un seul coup mais partie par partie.

```
Query query =
entityManager.createQuery("SELECT user FROM User As user");
query.setMaxResults(30);
query.setFirstResults(10);
List<User> listUsers = query.getResultList();
```

Dans cet exemple, nous récupérons au maximum 30 résultats à partir du 10e enregistrement.

Query setParameter(String parameterName, Object value) et Query setParameter(int parameterIndex, Object value)

Ces deux méthodes assignent la valeur `value` respectivement au paramètre `parameterName` ou au paramètre placé à l'index `parameterIndex`. Le nom du paramètre est défini dans la requête *via* « `:nomDuParametre` », ou *via* « `?numéro` ».

Les paramètres indexés sont placés *via* un point d'interrogation suivi d'un entier, dont la valeur commence par 0, et il peut être placé à plusieurs endroits simultanément.

```
SELECT user FROM User AS user WHERE user.id=?0
```

Les paramètres nommés sont placés grâce aux deux points suivis d'un identifiant.

```
SELECT user FROM User AS user WHERE user.id=:userId
```

Nous utilisons, bien entendu, les méthodes `setParameter()` de l'objet `Query` pour définir les paramètres à utiliser lors de l'exécution de la requête.

```
Query query = entityManager.createQuery(
"SELECT user FROM User AS user WHERE user.id=?0");
query = query.setParameter(0, new Integer(5));
```

Pour la requête basée sur les paramètres nommés, c'est un code similaire.

```
Query query = entityManager.createQuery(
"SELECT user FROM User AS user WHERE user.id=:userId");
query = query.setParameter("userId", new Integer(5));
```

Il existe également d'autres surcharges de la méthode `setParameter()`. En voici les détails :

- `Query setParameter (int position, Calendar value, TemporalType tempType)`
- `Query setParameter (String name, Calendar value, TemporalType tempType)`
- `Query setParameter (int position, Date value, TemporalType tempType)`
- `Query setParameter (String name, Date value, TemporalType tempType)`

Deux paramètres sont ajoutés, permettant d'y affecter une marque de temps, *via* `Date` ou `Calendar`, ainsi que le type de cette marque, *via* l'énumération `TemporalType` (valeurs possibles : `DATE`, `TIME`, `TIMESTAMP`).

Notez que ces méthodes retournent l'objet `Query` sur lequel elles travaillent. Cela n'est pas anodin et permet d'appeler ces méthodes en cascade, comme le montre l'exemple suivant :

```
Query query = entityManager.createQuery(
"SELECT user FROM User AS user WHERE user.id=:userId AND user.firstName=?0");
List<User> listUsers = query.setParameter("userId", new
Integer(5)).setParameter(0, "Alex").getResultList();
```

> **Remarque :** le principal avantage d'utiliser de telles requêtes « à paramétrer » est de pouvoir créer des requêtes génériques auxquelles le développeur pourra passer en paramètre tout type d'objet.

int executeUpdate()

Il est désormais possible d'exécuter des requêtes « UPDATE » (modification) ou « DELETE » (suppression). Pour cela, on utilise la méthode `executeUpdate()`.

Celle-ci exécute la requête qui doit être de type UPDATE ou DELETE. Elle retourne le nombre d'enregistrements supprimés ou modifiés.

```
Query query =
entityManager.createQuery("DELETE FROM User user WHERE user.login = '%s%'");
System.out.println(query.executeUpdate() + " enregistrement(s) supprimé(s)");
```

Nous supprimons, dans cet exemple, l'ensemble des utilisateurs dont le login contient un « s » puis nous affichons le nombre de suppressions effectuées.

Query setHint(String arg, Object arg)

Cette méthode permet d'utiliser des opérations spécifiques au fournisseur de persistance (*plugins*).

Il est possible, par exemple, de définir un *timeout* personnalisé avec Hibernate.

```
Query query = entityManager.createQuery("SELECT u FROM User u");
query.setHint("org.hibernate.timeout", 300);
```

Dans cet exemple, le temps maximal d'exécution d'une requête sera de 300 millisecondes.

7.3.3 Les requêtes nommées (*Named Queries*)

Les requêtes nommées peuvent s'apparenter aux « finders » en EJB 2. Contrairement aux requêtes dynamiques (vues précédemment), elles ont l'avantage de pouvoir être « précompilées » lors du déploiement et donc de s'exécuter plus rapidement.

Pour définir une requête nommée, il faut utiliser l'annotation `@javax.persistence.NamedQuery`. Celle-ci est définie par un nom et une requête EJB-QL associée.

```
@NamedQuery (
  name="findAllUserWithName"
  query="SELECT user FROM User user WHERE user.lastName LIKE :lastname" )
@Entity
public class User { ... }
```

Cette annotation doit être placée au niveau de la classe de l'Entity Bean. L'attribut `name` définit le nom de la requête et `query` la requête EJB-QL en elle-même. Vous pouvez remarquer l'utilisation d'une requête paramétrée dans l'exemple précédent.

Pour exécuter cette requête, il faut utiliser la méthode `EntityManager.createNamedQuery()`. Celle-ci prend en paramètre le nom de la requête nommée que l'on souhaite utiliser. L'Entity Manager fait automatiquement le lien avec les requêtes déclarées.

```
//...
@PersistenceContext
public EntityManager entityManager;

  public List<User> findUserByName(String name) {
    List<User> userList =
entityManager.createNamedQuery("findAllUserWithName").setParameter("lastname",na
me).getResultList();
    return userList;
  }
//...
```

Contrairement aux EJB 2, ces requêtes ne sont pas liées à des méthodes de l'Entity Bean. En effet, celles-ci n'ont plus d'interface « home » ni d'interface

« business ». Vous pouvez les utiliser dans tout objet disposant d'un Entity Manager (et principalement dans les Session Beans).

Si vous souhaitez définir plusieurs requêtes nommées pour un Entity Bean, il vous faudra les regrouper *via* l'annotation `@NamedQueries`.Voici quelques exemples de requêtes nommées qui pourraient être définies pour `User`.

```
@Entity
@Table(name = "User")
@NamedQueries( {
  @NamedQuery(
    name="findAllUsers",
    query="SELECT user FROM User AS user"),
  @NamedQuery(
    name="findUserByLogin",
    query="SELECT user FROM User AS user WHERE user.login=:login"),
  @NamedQuery(
    name="findUserByPrimaryKey",
    query="SELECT user FROM User AS user WHERE user.id=:id"),
  @NamedQuery(
    name="findUserByRange",
    query="SELECT user FROM User AS user WHERE user.id"
      +"BETWEEN :idMin AND :idMax")}
)
public class User { ... }
```

Voici la description des différentes requêtes de l'exemple :

- `findAllUsers` : récupère tous les Entity Beans `User`.
- `findUserByLogin` : récupère un Entity Bean ayant le `login` passé en paramètre. Sachant qu'il s'agit d'une propriété à valeur unique, la requête ne renverra qu'une instance.
- `findUserByPrimaryKey` récupère un Entity Bean à partir de sa clé primaire.
- `findUserByRange` pour retourner les Entity Beans ayant une clé primaire comprise entre `idMin` et `idMax`.

Attention : soyez attentif à ne pas avoir deux requêtes ayant le même nom. Une erreur sera générée au déploiement lors d'un tel conflit.

7.3.4 Les requêtes natives (*Native Queries*)

Une requête native est une requête écrite dans le langage natif de la base de données, c'est-à-dire SQL.

Les requêtes natives ont l'intérêt de pouvoir utiliser des fonctionnalités spécifiques à une base de données, et non disponibles dans EJB-QL (la commande `CONNECT` d'Oracle par exemple). Il s'agit également d'une solution intéressante lorsque l'on souhaite migrer une application de SQL vers EJB-QL. Bien sûr, il faut garder à

l'esprit que ces requêtes risquent de ne pas être portables si vous changez de base de données.

Remarque : les exemples suivants ont été testés avec MySQL 5.

La méthode `EntityManager.createNativeQuery()` permet de créer une requête native. Cette méthode retourne un objet `Query` comme les autres types de requêtes.

Voici un exemple de requête native permettant de récupérer tous les produits financiers d'un utilisateur donné :

```
Query query = em.createNativeQuery (
  "SELECT fp.* " +
  "FROM XUser user, Portfolio p, FinancialProduct fp " +
  "WHERE user.id=? AND user.id=p.userId " +
  "AND p.id=fp.portfolioId;"
);
query.setParameter(1, id);
Collection<FinancialProduct> collection = query.getResultList();
```

Quand cette requête est exécutée, elle retourne une collection d'instances de type `FinancialProduct`. On remarque que toutes les colonnes de la table « FinancialProduct » sont récupérées, afin de pouvoir recréer une instance avec toutes les informations nécessaires. L'oubli d'une seule colonne renvoie une exception de type `SQLException`.

Le passage de paramètres qui suit est similaire à celui des `PreparedStatement` de JDBC. De cette façon, nous devons placer des points d'interrogation « ? » pour chaque paramètre de la requête. Puis chacun doit être défini avant l'exécution de celle-ci, étant remplacés par les valeurs passées à l'objet `Query`.

Attention : les paramètres commencent par l'index 1.

Il est également possible de récupérer plusieurs entités différentes. Admettons pour cela que l'on souhaite récupérer une entité persistante de type `User`, et une autre de type `AccountInfo` qui s'y rapporte (pour rappel, ces deux entités ont une relation « One to One » entre elles). Dans ce cas-là, des *mappings* explicites entre la requête et les propriétés des Entity Beans doivent être mis en place grâce à l'annotation `@SqlResultSetMapping`.

```
@Entity
@Table(name = "XUser")
@NamedNativeQuery(name="findUserInformations",
  query="SELECT user.* , accountInfo.* " +
  "FROM XUser user " +
  "LEFT OUTER JOIN AccountInfo accountInfo ON " +
  "user.id = accountInfo.id",
   resultSetMapping="userInformations")

@SqlResultSetMapping(
  name="userInformations",
  entities= {
```

```
  @EntityResult(entityClass=
    com.labosun.stockmanager.entity.User.class),
  @EntityResult( entityClass=
    com.labosun.stockmanager.entity.AccountInfo.class)
})

public class User { ... }
```

Dans l'exemple précédent, nous définissons un *mapping* nommé `userInformations` contenant les Entity Beans `User` et `AccountInfo`.

Voici une description de l'annotation `@SqlResultSetMapping` :

- `name` : définit le nom du *mapping* SQL.
- `entities` : déclare les différents Entity Beans intervenant dans la requête. On utilise pour cela un tableau d'`EntityResult`.
- `columns` : déclare les *mappings* à utiliser pour les valeurs scalaires (utilisé lorsque la requête ne renvoie pas que des Entity Beans, mais des valeurs annexes).

Le paramètre `resultSetMapping` de l'annotation `@NamedNativeQuery` permet de relier la requête avec le *mapping* approprié. Les autres sont similaires aux paramètres de l'annotation `@NamedQuery`.

Voici une description de l'annotation `@EntityResult` :

- `entityClass` : définit la classe de l'Entity Bean utilisé (objet de type `Class`).
- `fields` : définit la liaison entre les champs retournés par la requête et les propriétés de l'Entity Bean. Il faut utiliser un tableau de `@FieldResult`.
- `discriminatorColumns` : définit la colonne (ou l'alias) permettant de distinguer le type de l'Entity Bean d'une même hiérarchie (définie par un héritage).
- `fields` : permet de *mapper* les champs d'une requête aux propriétés des Entity Beans récupérés. Pour cela, on utilise un tableau de `@FieldResult`.

L'appel d'une requête native définie en tant que requête nommée, ne change guère avec l'appel d'une requête nommée (type EJB-QL).

```
Query query = em.createNamedQuery("findUserInformations") ;
Collection<Object[]> informationCollection = query.getResultList() ;
for(Object[] o : informationCollection){
  // o[0] est un entity bean de type User
  // o[1] est un entity bean de type AccountInfo
}
```

Dans l'exemple précédent, nous récupérons à la fois une instance `User` et une instance `AccountInfo` à chaque itération, comme cela est défini dans la requête et le *mapping*.

La syntaxe se complique un peu si l'on souhaite renommer le nom des champs de la requête (en utilisant des alias). Dans ce cas, il faut relier chaque colonne avec sa

propriété de l'Entity Bean liée. L'exemple suivant effectue le *mapping* des champs de l'entité `AccountInfo` pour la requête :

```
SELECT ai.id AS id, ai.cardNumber AS cardNumber, ai.amount AS amount,
ai.accountId AS accountId
```

Voici l'`EntityResult` associé :

```
@EntityResult (
entityClass=com.labosun.stockmanager.entity.AccountInfo.class,
fields={
  @FieldResult(name="id", column="id"),
  @FieldResult(name="cardNumber", column="cardNumber"),
  @FieldResult(name="amount", column="amount"),
  @FieldResult(name="accountId", column="accountId"),
})
```

Même si l'utilisation de requêtes natives n'offre pas une portabilité sûre de votre application, elle offre la possibilité d'optimiser certaines requêtes en fonction de la base de données.

Il est cependant impossible d'appeler directement des procédures stockées. Une solution est de les appeler à partir d'une requête SELECT ou *via* une implémentation propriétaire (par exemple, Hibernate propose l'objet `org.hibernate.Callable`). Ceci étant une fonctionnalité spécifique, nous ne détaillons pas son fonctionnement.

Remarque : pour être le plus portable possible, la requête ne doit utiliser que des paramètres positionnés avec des « ? ». Il est également possible d'utiliser des paramètres nommés (`:id`, `:lastname`...) mais cela est déconseillé. En effet, les deux points peuvent êtres confondus, dans certains cas, avec ceux utilisés dans les *triggers* et les procédures stockées.

En résumé

L'EJB-QL des EJB 3 est désormais complet. Forte de sa portabilité et de son niveau d'abstraction, cette nouvelle version de l'EJB-QL permet une optimisation complète du système de mapping objets/relationnel.

8

Développement des clients

Objectif

Nous avons principalement parlé, jusque-là, du développement de la partie serveur d'une application orientée entreprise. Mais il ne faut pas oublier que le ou les clients font, eux aussi, partie intégrante de la solution.

Après une présentation des concepts généraux du développement d'applications clientes, nous présenterons les différentes possibilités avec EJB 2 et EJB 3. Nous terminerons par le développement d'une application *standalone* J2SE.

8.1 CONNEXION CLIENT/SERVER

Cette partie présente les différents types de clients qui peuvent être mis en place pour se connecter aux EJB. Les différents principes expliqués ici sont communs aux spécifications EJB 2 et EJB 3.

8.1.1 Les différents clients

Les EJB étant des composants dit « serveur », un client doit obligatoirement être mis en place pour pouvoir communiquer avec ceux-ci. Ces clients peuvent prendre plusieurs formes : une application avec ou sans interface graphique, une applet, une servlet dans une application web ou un EJB.

Remarque : vous pouvez également appeler un EJB dans une page JSP. Néanmoins, cette technique est déconseillée car elle ne respecte pas les bons principes du pattern MVC (Modèle vue contrôleur). Ce modèle exige que l'accès au Modèle (service) se fasse depuis le Contrôleur (une servlet par exemple) et non à partir de la Vue (JSP).

8.1.2 Principe général

Le client localise l'EJB qu'il souhaite récupérer *via* l'API JNDI et le nom JNDI de cet EJB. Les composants déployés sont enregistrés dans l'annuaire du serveur. Seul le pilote JNDI, appelé *service provider*, change d'une implémentation à l'autre.

Les appels de méthodes distantes se font par RMI (*Remote Method Invocation*) alors que les appels de méthodes locales se font directement dans la JVM du serveur.

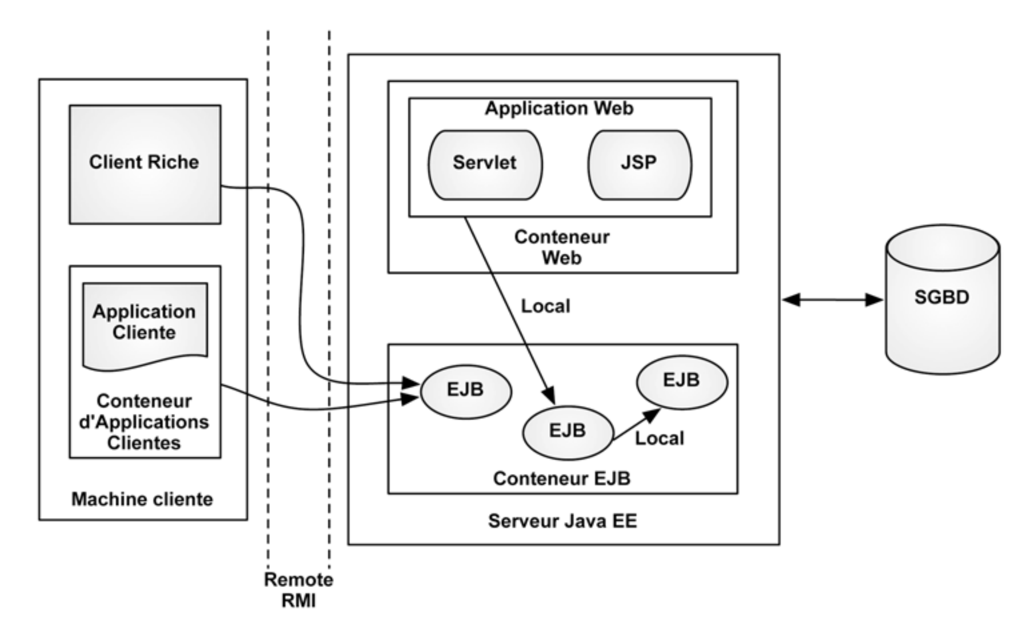


Figure 8.1 — Schéma général de communication client/EJB

Le client récupère une référence (implémentant l'interface métier) de l'EJB qu'il souhaite utiliser (fig. 8.1). Celui-ci peut alors appeler les méthodes de l'objet récupéré sans se soucier des contraintes de communication. En effet, l'appel d'une méthode est automatiquement transmis à l'instance de l'EJB dans le conteneur (généralement par un système de *proxy*). Cette instance traite la méthode et retourne le résultat au client. La création du *proxy* est à la charge du conteneur et reste totalement transparente pour le client.

8.1.3 Client local et conteneur

Un client ne communique pas nécessairement avec le serveur via le réseau. Au contraire, dans une grande partie des applications utilisant les EJB, les clients sont locaux. Un composant EJB ne travaillant jamais seul, il est commun d'utiliser un Session Bean pour regrouper la logique métier d'un cas d'utilisation. Celui-ci peut alors dépendre d'autres Beans ou ressources diverses.

Un client local, par définition, se situe dans la même JVM que celle du serveur d'applications. De ce fait, nous retrouvons les clients locaux dans les conteneurs WEB ou EJB. Même s'il est possible de définir des EJB accessibles à distance, il est conseillé de préférer la vue locale tant que possible. En effet, les accès locaux entre les composants sont beaucoup plus performants qu'à distance.

La localisation d'un EJB au sein d'un conteneur est assez spéciale car vous devez déclarer des références vers cet EJB dans le descripteur de déploiement de l'application. Vous utilisez alors la référence déclarée plutôt que le nom JNDI de l'EJB. Même s'il est possible d'accéder aux EJB *via* leur nom JNDI, nous verrons qu'il est préférable d'utiliser des références vers ces EJB. Le programme reste alors le plus indépendant possible, et, si des changements interviennent sur le nom de l'EJB, les clients ont juste à modifier les références dans leur descripteur de déploiement. Le code reste inchangé et aucune nouvelle compilation n'est alors nécessaire.

8.2 CLIENTS EJB 2

8.2.1 Présentation générale

L'utilisation d'un EJB 2 suit toujours la même logique (quel que soit le type de client) :

- Obtenir une référence implémentant l'interface home de l'EJB *via* JNDI. On appelle communément cette référence : la fabrique (*factory*).
- Créer une instance de l'interface métier (distante ou locale) grâce à l'objet précédemment récupéré (en appelant la méthode `create()`).
- Appeler les méthodes de l'EJB.

Voici un exemple illustrant la connexion d'un client distant à l'EJB `RichClientService` :

```
Context ctx;
try {
  // crée un contexte JNDI avec les valeurs du fichier jndi.properties
  // (qui doit se trouver à la racine du classpath)
  ctx = new InitialContext();

  // récupère la fabrique d'EJB "RichClientService" (home interface)
  Object home = ctx.lookup("ejb/RichClientService");

  RichClientServiceHome clientServiceHome = (RichClientServiceHome)
    PortableRemoteObject.narrow(home, RichClientServiceHome.class);
  RichClientService richClientService = clientServiceHome.create();
} catch (Exception e) {
  // gestion des erreurs
}
```

L'utilisation de la méthode `PortableRemoteObject.narrow()` est obligatoire pour les clients distants. En effet, la méthode `lookup()` de l'objet `Context` retourne une instance de `Object` qui doit être *castée* (changement de type forcé) avec l'interface home de l'EJB. Malheureusement, il n'est pas possible d'utiliser un *cast* explicite dans ce cas-là.

```
// impossible pour les clients distants
RichClientServiceHome clientServiceHome = (RichClientServiceHome)
  ctx.lookup("ejb/RichClientService");
```

Cette norme est définie afin de garantir la compatibilité d'EJB avec Corba.

La communication EJB entre le client et le serveur est basée sur RMI. Toutefois, le protocole sous-jacent utilisé est IIOP. Celui-ci n'a pas été écrit pour Java précisément, mais de façon générique (pour tous les langages), ce qui induit certaines limitations. Par exemple, certains langages n'intègrent pas le concept de *casting*. Par conséquent, dans le cas des accès distants, en java, vous êtes obligé de vérifier le type de l'objet récupéré par la méthode `lookup()` *via* `PortableRemoteObject.narrow()`.

Remarque : il est tout à fait possible que, dans certains cas, le *cast* explicite ne pose pas de problème. Cependant, si vous souhaitez opter pour un code standard et portable, il est conseillé de respecter au maximum les spécifications.

8.2.2 Dans les conteneurs EJB et web

Comme nous venons de le voir, un client peut accéder à un EJB par le nom JNDI de celui-ci.

```
//...
protected CommonServiceHome getCommonServiceHome() throws Exception {
  Context context = new InitialContext();
  return (CommonServiceHome) context.lookup("ejb/CommonService");
}
//...
```

Nous accédons, localement à l'EJB dont le nom JNDI est : « ejb/CommonService ».

Cependant, lorsque le client se trouve dans un conteneur, il est recommandé d'utiliser des références, qui doivent être paramétrées dans le descripteur de déploiement de l'application.

Dans le cas du conteneur EJB, les références entre les EJB sont définies dans le fichier « ejb-jar.xml ».

Attention : les références ne sont pas globales à l'application, mais doivent être définies pour chaque EJB.

Voici une définition de référence vers l'EJB CommonService pour l'EJB RichClientService. Pour cela, on utilise la balise <ejb-local-ref> ou <ejb-ref> dans le fichier « ejb-jar.xml ».

```
<ejb-jar version="2.1">
<session>
  <ejb-name>RichClientService</ejb-name>
  <ejb-local-ref>
    <ejb-ref-name>CommonService</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>
      com.labosun.stockmanager.ejb2.session.interfaces.CommonServiceLocalHome
    </local-home>
    <local>
      com.labosun.stockmanager.ejb2.session.interfaces.CommonServiceLocal
    </local>
    <ejb-link>CommonService</ejb-link>
  </ejb-local-ref>
</session>
</ejb-jar>
```

Remarque : la balise <ejb-local-ref> permet d'établir une référence vers un EJB situé dans le même JAR que celui détenant cette référence. Si vous souhaitez accéder à un EJB situé dans un autre JAR du même serveur, vous devrez utiliser la balise <ejb-ref>.

Vous pouvez remarquer que la balise <ejb-local-ref> est incluse dans la balise <session>. De ce fait, la référence n'est utilisable que pour le Bean dans laquelle elle est déclarée.

La récupération de l'objet *home* de l'EJB s'établit, alors, de la façon suivante :

```
//...
protected CommonServiceLocalHome getCommonServiceHome() throws Exception {
  Context context = new InitialContext();
  return (CommonServiceLocalHome) context.lookup("java:comp/env/CommonService");
}
```

La recherche JNDI s'opère alors sur la référence locale dont le nom est défini par la balise <ejb-link>. La liaison vers le nom JNDI de l'EJB est faite automatiquement par l'API JNDI.

Dans le cas du conteneur web, les références sont définies dans le fichier « web.xml ». Contrairement au fichier « ejb-jar.xml » qui impose que les références soient liées à un EJB précis, le fichier « web.xml » permet l'utilisation de références globales à l'application.

Les balises utilisées (<ejb-ref> et <ejb-local-ref>) sont les mêmes que celles utilisées dans le descripteur de déploiement EJB. Voici un exemple :

```
<web-app version="2.4">
...
 <ejb-ref>
    <ejb-ref-name>RichClientService</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
```

```
      <home>
        com.labosun.stockmanager.ejb2.session.interfaces.RichClientServiceHome
      </home>
      <remote>
        com.labosun.stockmanager.ejb2.session.interfaces.RichClientService
      </remote>
      <ejb-link>ejb/RichClientService</ejb-link>
    </ejb-ref>
...
</web-app>
```

Grâce à cette déclaration, vous pouvez utiliser le nom JNDI « java:comp/env/RichClientService » pour accéder à l'EJB référencé. L'appel peut aussi bien être fait dans une servlet, JSP ou même un JavaBean.

8.3 CLIENTS EJB 3

8.3.1 Présentation générale

Une fois de plus, la spécification a simplifié le code du client. L'interface *home* ayant été « supprimée », le client récupère directement une référence (implémentant l'interface métier de l'EJB) avec JNDI.

Voici un exemple permettant à un client de récupérer une référence vers l'EJB `RichClientServiceBean`.

```
try {
  Context ctx = new InitialContext();
  // le nom JNDI par défaut est
  // "Nom du fichier EAR/Nom de la classe du bean/type accès : local/remote"
  richClientService = (RichClientService)
    ctx.lookup("StockManager/RichClientServiceBean/remote");
} catch (NamingException e) {
  // gestion des erreurs
}
```

Le principe est le même que ce soit un client dans une application web, une application console ou un autre EJB. Même si le code a été réduit de moitié entre la spécification EJB 2 et EJB 3, là n'est pas la grande nouveauté.

Il est dorénavant possible de gérer les dépendances des clients vis-à-vis des EJB par simple injection. Comme vu précédemment avec l'Entity Manager (chapitre 6), vous pouvez préciser au conteneur que votre EJB est dépendant de tel autre. Le conteneur se chargera d'injecter automatiquement l'instance demandée. Pour cela, il suffit d'annoter la propriété avec **@EJB**. Voici un exemple utilisant cette annotation :

```
@Stateful
@Remote( { RichClientService.class })
public class RichClientServiceBean implements RichClientService {
//...
```

```
  @EJB
  private CommonService commonService;
//...
}
```

La classe `RichClientServiceBean` définit une variable d'instance `commonService` dont le type est `CommonService` (l'interface métier liée à l'EJB). Grâce à l'annotation `@EJB` positionnée sur cette propriété, le conteneur injecte automatiquement une instance de l'EJB (`CommonServiceBean`) lors de l'instanciation de l'EJB `RichClientServiceBean`. C'est-à-dire qu'il fait, à votre place, la recherche JNDI et l'initialisation de la dépendance vers cet EJB.

Pour certains cas plus complexes, l'annotation `@EJB` admet des attributs qui permettent de préciser la localisation de l'EJB.

- `beanName` : spécifie le nom de l'EJB (spécifié par l'attribut `name` des annotations `@Stateless`, `@Stateful` ou dans le descripteur de déploiement avec la balise `<ejb-name>`).
- `beanInterface` : spécifie l'interface métier que l'on souhaite utiliser. Cet attribut est principalement renseigné lorsque vous utilisez une interface parente de l'interface métier implémentée par le Bean. L'interface par défaut est celle utilisée par la variable qui est sujette à l'injection.
- `mappedBy` : spécifie le nom JNDI à utiliser pour rechercher l'instance du Bean.

La spécification EJB 3 souhaite cacher ces appels JNDI qui ont souvent dérouté les développeurs. Toutefois, l'injection a ses limites et n'est disponible qu'au sein d'un conteneur.

Attention : l'annotation `@EJB` dans une application client riche (console ou graphique) n'est disponible que lorsque celle-ci est lancée dans l'*Application Client Container* (AAC). La mise en place de ce type d'application est expliquée ci-après.

8.3.2 Dans les conteneurs EJB et web

La spécification EJB 3 intégrant le principe d'injection, il n'est plus nécessaire de définir les références dans le fichier « ejb-jar.xml » ou « web.xml ». En effet, le conteneur détecte les annotations `@EJB` et gère les dépendances automatiquement.

Toutefois, il existe certaines restrictions. Dans le conteneur EJB, seuls les Session Beans et les Message Driven Beans peuvent utiliser l'injection. Dans le conteneur web, seuls les composants « Servlet » et « JSF Managed Bean » peuvent utiliser l'injection.

8.4 APPLICATION CLIENT CONTAINER

Il est possible de créer un client Java SE, dit *standalone*, afin de travailler avec des EJB accessibles à distance. Il existe, pour cela, un conteneur client spécifique qui offre beaucoup d'avantages pour la mise en œuvre de ces clients distants. C'est le conteneur d'application cliente (ACC, *Application Client Container*). L'usage du conteneur client n'étant pas exclusivement réservé aux EJB, nous présenterons cet élément de façon pragmatique. En effet, les objectifs de cette partie sont de vous familiariser avec ce conteneur et de vous présenter les nouveautés apportées par Java EE.

8.4.1 Présentation

Le conteneur d'application client inclut un ensemble de classes java, de librairies, et d'autres fichiers requis. Cet ensemble est généralement fourni avec le serveur d'applications et ses dépendances sont distribuées avec le programme client java qui s'exécute dans sa propre machine virtuelle (JVM), sur le poste client.

Le conteneur gère alors l'exécution du programme client et offre l'accès à de nombreux services Java EE, disponibles sur le serveur d'applications, *via* le protocole RMI-IIOP. Si on le compare avec les autres conteneurs (EJB, WEB), on peut le caractériser aussi de « conteneur léger ».

Voici les avantages que ce conteneur apporte :

- L'application a accès à un espace JNDI local « java:comp/env/ » comme les composants Java EE. Cela induit que les EJB ou autres ressources peuvent être déclarés et placés dans cet espace JNDI. Les liens ou références peuvent alors être résolus vers le serveur actuel en utilisant un descripteur de déploiement.
- Alors que seuls les EJB sont accessibles à distance dans une application *standalone*, le conteneur client offre la possibilité d'obtenir des références vers des connecteurs Java EE, des pools de connexions ou des ressources JMS.
- Le conteneur peut gérer le processus de login et fournir une connexion sécurisée (SSL) de façon totalement transparente pour le développeur.

De façon plus générale, l'utilisation du conteneur client offre une meilleure portabilité à vos applications. De plus, le code est plus évolutif car les liaisons avec le serveur sont regroupées dans les fichiers .xml de description.

8.4.2 Paramétrage

Comme tout module Java EE, une application cliente est représentée par un fichier .jar contenant les classes du programme.

Les applications clientes compatibles avec la spécification J2EE 1.4 doivent obligatoirement définir un fichier « application-client.xml » qui correspond au descrip-

teur de déploiement de l'application. Ce fichier doit se trouver dans le dossier « META-INF » du .jar client. Il est lu par le conteneur avant le lancement de l'application.

Voici un exemple de fichier « application-client.xml » :

```
<?xml version="1.0" encoding="UTF-8"?>
<application-client xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/application-client_1_4.xsd"
  version="1.4">

  <display-name>Nom pour l'application cliente</display-name>
  <description>Description pour l'application cliente</description>

  <env-entry>
    <env-entry-name>nameEntry</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>
       Déclaration d'une entrée dans l'environnement de type String
    </env-entry-value>
  </env-entry>

  <ejb-ref>
    <ejb-ref-name>ejb/RichClientService</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>
      com.labosun.stockmanager.ejb2.session.interfaces.RichClientServiceHome
    </home>
    <remote>
      com.labosun.stockmanager.ejb2.session.interfaces.RichClientService
    </remote>
    <ejb-link>RichClientService</ejb-link>
  </ejb-ref>
</application-client>
```

Cet exemple définit une entrée d'environnement `nameEntry` qui pourra être utilisée dans l'application cliente. Ce client pourra accéder à l'EJB « ejb/ RichClientService » par l'intermédiaire de la référence « RichClientService ». Cette référence a été déclarée par la balise `<ejb-ref>`. Toutefois, nous ne nous attardons pas sur ce fichier qui est facultatif depuis la spécification Java EE 5.

Dans tous les cas, il faut définir la classe de démarrage qui sera exécutée par le conteneur. Pour cela, il suffit de la spécifier dans le fichier « manifest.mf » qui est présent dans le dossier « META-INF » du .jar client.

Voici un exemple de « manifest.mf » :

```
Main-Class: com.labosun.stockmanager.client.application.ApplicationClientMain
```

Attention : n'oubliez surtout pas de laisser une (ou plusieurs) ligne(s) vide(s) à la fin de ce fichier.

La classe de démarrage est définie par l'attribut « Main-Class ». Dans notre exemple, le conteneur lancera la méthode main de la classe `com.labosun.stockmanager.client.application.ApplicationClientMain`.

8.4.3 Utilisation et exécution sous Java EE 5

Contrairement aux clients *standalone*, un client « container-managed » peut utiliser l'injection pour récupérer des références vers les EJB dont il dépend. Cela évite non seulement l'écriture de la localisation JNDI mais aussi le paramétrage du fichier « application-client.xml ». Les références vers les EJB sont automatiquement détectées et gérées par le conteneur client.

Injection d'EJB

L'utilisation de l'annotation `@EJB` dans une application cliente diffère par rapport à son utilisation au sein du conteneur EJB. L'annotation ne peut être utilisée qu'à l'intérieur même de la classe de démarrage (« Main-Class » identifiée dans le « manifest.mf ») et doit être positionnée sur une propriété `static`. Cela est dû au fait que le conteneur exécute la méthode `main()` qui est elle-même `static`.

Voici un exemple d'une classe de démarrage :

```
public class ApplicationClientMain {
  // la propriété clientService est "static"
  @EJB
  private static RichClientService clientService;

  public static void main(final String... args) {
    // utilisation de clientService initialisée par le conteneur
  }
}
```

Il est également possible d'injecter d'autres ressources (destination JMS, entrée d'environnement …) *via* l'annotation `@Resource`.

```
@Resource(mappedName="jms/TestQueue") private static Queue testQueue;
```

Il est cependant impossible d'injecter une référence vers un EJB dans une variable d'instance. En effet, le conteneur ne crée pas d'instance de la classe `ApplicationClientMain`, mais exécute directement la méthode `main()`.

> **Remarque :** lorsque la méthode `main` de la classe cliente se termine, certains conteneurs quittent la JVM automatiquement. Cela peut être dérangeant dans le cas d'une application graphique (la fenêtre s'ouvre et le programme quitte aussitôt). Pour remédier à cela, vous pouvez utiliser une boucle infinie à la fin de la méthode `main`.

Execution

L'application cliente s'exécute dans un conteneur. De ce fait, c'est ce dernier qui démarre l'application et non la machine virtuelle directement (comme pour les

applications Java *standalone*). Les commandes permettant de lancer le conteneur ne sont pas standardisées et sont donc dépendantes du fournisseur utilisé.

Glassfish propose le script « appclient.bat » (pour Windows) pour lancer le .jar d'application client. Pour cela vous devez le spécifier en argument `-client` :

```
appclient -client C:\chemin\vers\lefichier\client\ClientApp.jar
```

Voici un rapide aperçu du script « appclient.bat » :

```
@echo off
REM
REM Copyright 2004-2005 Sun Microsystems, Inc. All rights reserved.
REM Use is subject to license terms.
REM

setlocal
call "C:\glassfish\config\asenv.bat"

rem environment set by tools
if X%TOOLS_SETTINGS% == X goto skip
   set AS_INSTALL=%S1AS_INSTALL_ROOT%
   set AS_JAVA=%S1AS_JAVA_HOME%
   set AS_ACC_CONFIG=%S1AS_ACC_CONFIG%
   set AS_IMQ_LIB=%S1AS_IMQ_LIB%
   set AS_WEBSERVICES_LIB=%S1AS_WEBSERVICES_LIB%
:skip
...
set JVM_CLASSPATH=%AS_INSTALL%\lib\appserv-rt.jar;
  %AS_INSTALL%\lib\javaee.jar;
  %AS_INSTALL%\lib\appserv-ext.jar;
  %WEBSERVICES_CLASSPATH%;
  %AS_INSTALL%\lib\install\applications\jmsra\imqjmsra.jar;
  %AS_IMQ_LIB%\fscontext.jar;
  %AS_INSTALL%\lib\appserv-cmp.jar;
  %AS_DERBY_INSTALL%\lib\derbyclient.jar;
  %AS_INSTALL%\lib\toplink-essentials.jar;
  %AS_INSTALL%\lib\dbschema.jar;
  %AS_INSTALL%\lib\appserv-admin.jar;
  %AS_INSTALL%\lib\dtds;
  %AS_INSTALL%\lib\schemas;
...
  "%AS_JAVA%\bin\java" %ACTIVATION% %VMARGS%
  -Djava.util.logging.manager=com.sun.enterprise.server.logging.ACCLogManager
  com.sun.enterprise.appclient.Main -configxml "%AS_ACC_CONFIG%" %*

endlocal
```

Le script précédent opère principalement sur les variables d'environnement. Ces variables configurent, en grande partie, le classpath à utiliser pour le lancement du conteneur. La variable principale est JVM_CLASSPATH et regroupe l'ensemble des librairies utilisées par le conteneur. Celui-ci est lancé par la commande `java` qui exé-

cute la classe passée en paramètre (ici `com.sun.enterprise.appclient.Main`) dans la machine virtuelle.

Malgré les avantages qu'un conteneur apporte à une application cliente, son utilisation s'avère délicate. Le système qui exécute l'application cliente doit posséder l'ensemble des librairies du conteneur et le script de lancement. De plus, si des modifications sont apportées au client (correction de bugs, évolution...), les binaires de l'application ne seront pas mis à jour automatiquement. Ces nombreux problèmes ont souvent été un frein pour les entreprises qui ont été dans l'obligation de trouver des solutions annexes.

Java Web Start (JWS)

Nous parlions, dans la partie précédente, des problèmes de maintenance et de déploiement d'une application riche côté client. C'est à partir de ces problèmes qu'un standard a émergé : *Java Web Start* (JWS).

Cette solution apporte les avantages suivants :

- Accès à l'application cliente *via* une URL (navigateur internet).
- Garantite de toujours exécuter la dernière version de l'application *via* un système d'*auto-update* (mises à jour automatiques).
- Épargne les procédures d'installation ou de mise à niveau compliquées.
- Portable (intégration dans la machine virtuelle).

Il est donc possible d'utiliser cette technologie pour faciliter le déploiement des applications clientes. Toutefois, la spécification Java EE 5 n'impose pas son utilisation et son intégration dans un serveur d'applications. Le choix de son implémentation est donc laissé au fournisseur de celui-ci.

Le projet « Glassfish » intègre cette fonctionnalité. L'activation de celle-ci se fait lors du déploiement grâce à l'interface d'administration (fig. 8.2).

L'exécution de l'application cliente peut se faire directement *via* l'URL de déploiement de celle-ci par JWS, par exemple :

```
http://www.monserveur.com:8080/StockManager/StockManagerRichClientApp.jnlp
```

Mais elle peut également être exécutée *via* la commande « javaws » suivie de l'URL de déploiement (fig. 8.3) :

```
javaws http://192.168.0.100:8080/StockManager/StockManagerRichClientApp.jnlp
```

JWS offre de nombreux avantages pour le déploiement d'applications clientes riches au sein d'un parc informatique : il permet d'auto-configurer le conteneur client avec les paramètres du serveur de production et tout cela sans le moindre effort.

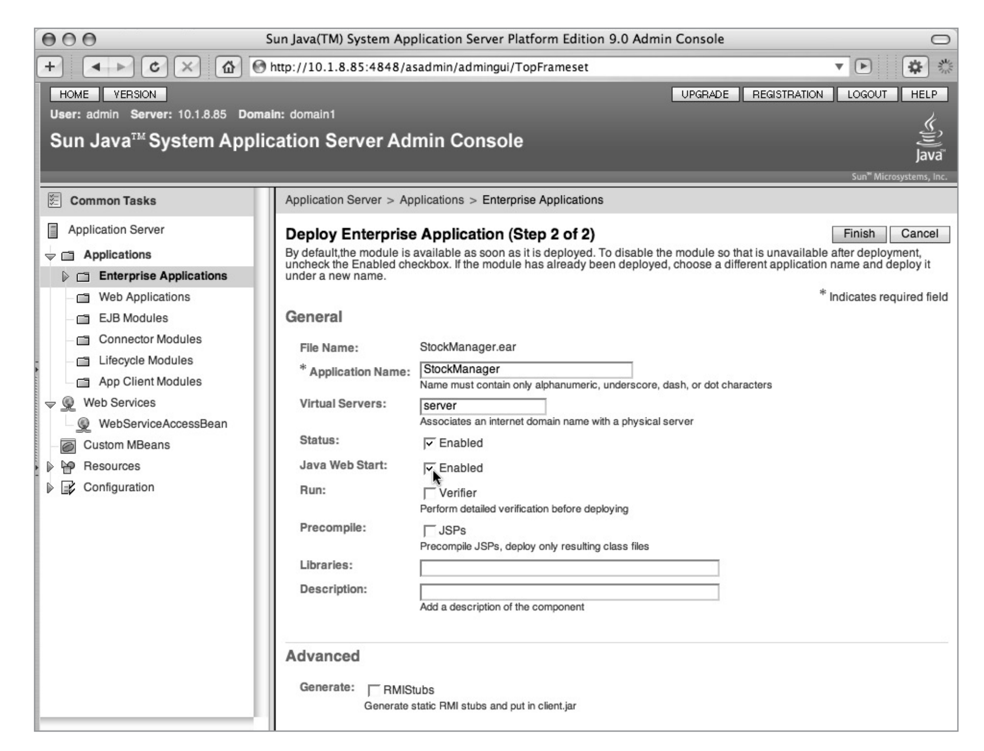


Figure 8.2 — Déploiement d'une application cliente avec GlassFish et JWS



Figure 8.3 — Démarrage de l'application *via* JWS

En résumé

Le développement d'applications clientes, grandement facilité par les EJB 3, est désormais à son apogée. La simplicité du développement de clients riches connectés aux EJB et mis à jour automatiquement y est pour quelque chose. *Java Web Start*, qui tend à être un standard, n'est cependant pas encore très implanté sur les postes clients.

9

Les transactions

Objectif

L'utilisation du système de persistance EJB implique l'utilisation d'un système transactionnel important. Que les clients soient locaux ou distants, même constat : les différentes actions métiers effectuées sur les données doivent répondre de la cohérence de celles-ci.

Il est alors important, dans la conception de l'application, de prendre en compte les différents niveaux de transactions.

Nous étudierons, dans un premier temps, le concept des transactions au sein d'une architecture Java EE, puis nous l'appliquerons ensuite aux EJB 2 et 3.

9.1 LE MODÈLE TRANSACTIONNEL

Le concept de transaction se retrouve aujourd'hui dans la majeure partie des applications gérant une persistance de données.

Lorsqu'il existe des relations ou des contraintes entre les données de l'application, nous devons être sûrs du résultat des différentes opérations. Il faut être certain qu'une suite d'opérations s'est déroulée correctement du début à la fin, et, dans la négative, rétablir les données d'origine.

La transaction est une solution permettant de regrouper une série d'opérations dans un tout indivisible. Cette série est alors « validée » ou « annulée ».

Une transaction est souvent identifiée par l'acronyme ACID :

- *Atomicity* (atomicité) : réponse binaire (validée ou annulée).
- *Consistency* (cohérence) : les données sont toujours valides.
- *Isolation* : la vision des données au sein même de la transaction reste indépendante vis-à-vis des autres transactions.
- *Durability* (durabilité) : les résultats de ces opérations sont rendus persistants en base de données.

On retrouve généralement les transactions dans les systèmes bancaires, de ventes en ligne, de billetteries…

La gestion des transactions se complexifie en fonction du nombre de ressources différentes à gérer. Si l'application utilise une source de données unique, la gestion des transactions sera alors simple. Au contraire, si l'application intègre de nombreuses sources de données, les transactions devront être multipliées et regroupées dans un gestionnaire de transactions.

En Java EE, les applications sont déployées dans les conteneurs intégrés aux serveurs d'applications. C'est par ce dernier que se fera l'accès aux sources de données.

Deux acteurs, permettant la gestion des transactions, existent au niveau des serveurs d'applications :

- Le « *Resource Manager* » (RM) : le gestionnaire de ressources est un adaptateur entre l'application et le système de données. C'est lui qui s'occupe de traiter les instructions qui lui sont envoyées et d'assurer le maintien en cohérence des données sur le système de données. Celui-ci pouvant être un système de persistance (Base de données), un connecteur vers un autre système (ERP[1]) ou encore un annuaire (LDAP).
- Le « *Transaction Manager* » (TM) : le gestionnaire de transactions s'assure que l'ensemble des instructions a bien été exécuté. C'est lui qui vérifie la cohérence des informations pour valider les modifications (COMMIT) ou, au contraire, effectuer un retour en arrière en cas d'erreurs (ROLLBACK).

9.1.1 Les transactions locales et globales

Une transaction peut rassembler un ou plusieurs gestionnaires de ressources. Sa gestion sera affectée par le choix du nombre de gestionnaires utilisés pour la traiter.

Prenons l'exemple d'une série d'opérations agissant sur une même source de données. Nous avons ici un cas où la transaction utilise un seul gestionnaire de ressources. Nous pouvons alors regrouper l'ensemble des ordres SQL en une seule transaction (gérée par JDBC). On parle alors de transaction SQL que le SGBDR gérera (fig. 9.1).

1. ERP (*Entreprise Resource Planning*) : système intégré qui optimise les processus de gestion d'une entreprise. De la comptabilité à la gestion en passant par les managers ou les ressources humaines, toutes les ressources sont partagées.

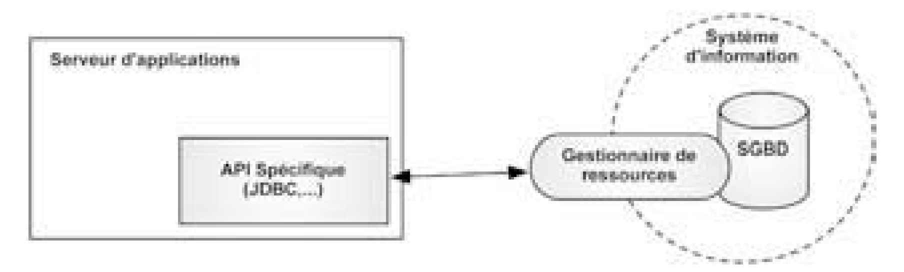


Figure 9.1 — Transaction au sein d'une source de données JDBC

Dans ce cas, le gestionnaire de ressources (RM) fait également office de gestionnaire de transactions (TM). On parle de « transaction locale au gestionnaire de ressources. »

À présent, si une transaction rassemble plusieurs gestionnaires de ressources, il convient alors de reconsidérer les transactions de façon globale.

La norme JTA/XA (*Java Transaction API*) de Java EE a été développée pour résoudre ce problème. Celle-ci permet à chaque gestionnaire de ressources d'utiliser un gestionnaire de transactions externes implémentant l'API JTA et utilisant un protocole d'échange standard, XA.

C'est par l'implémentation de l'interface standard XAResource qu'il sera possible de mettre en place un tel gestionnaire de ressources (fig. 9.2).

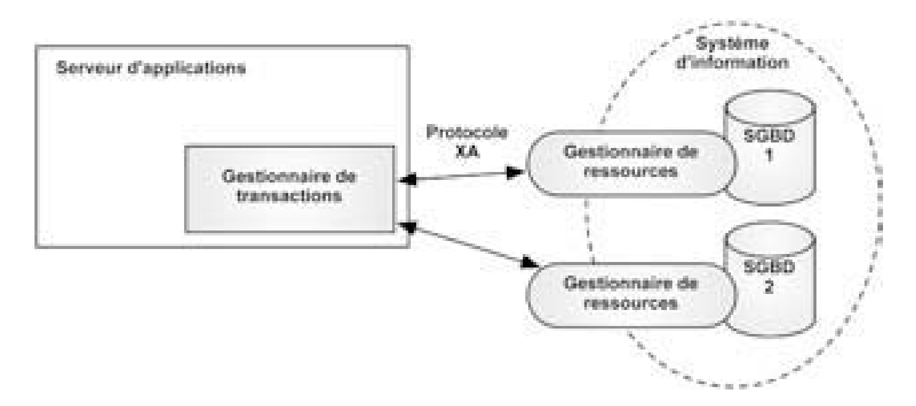


Figure 9.2 — Schéma de connexion APP <> TM <> RM

Ici, le gestionnaire de transactions (TM) fait l'intermédiaire entre l'application et le gestionnaire de ressources (RM). On parle de transaction globale au gestionnaire de ressources.

Remarque : par la suite, le gestionnaire de ressources sera assimilé à un SGBD (Système de gestion de bases de données), ce qui est généralement le cas dans la plupart des applications.

9.1.2 Les transactions concurrentes : niveaux d'isolation

Une transaction est dite « atomique » lorsque tout est validé ou tout est annulé. Un SGBD accepte généralement plusieurs connexions qui peuvent tout à fait lire, modifier ou supprimer des données simultanément.

Cependant, plusieurs transactions peuvent travailler sur les mêmes informations, à des moments différents, avant que l'une ou l'autre ne soit terminée. Il est alors possible que la cohérence des données puisse être perdue à cause d'un problème de concurrence d'accès.

Les trois problèmes de cohérence les plus communs sont :

- « *dirty reads* » : une « lecture sale » se produit lorsqu'une donnée, qui est lue par une connexion, subit un *rollback* d'une autre connexion. La donnée lue n'est alors plus cohérente avec sa valeur réelle en base (fig. 9.3).

  Par exemple, supposons que `account1.debit()` est appelée pour soustraire un montant au compte `account1` en parallèle, un autre processus appelle `account1.getSolde()` (cette méthode retourne alors un solde nul car le compte est épuisé). Le premier processus appelle maintenant `account2.credit()` qui échoue et provoque un *rollback* complet de la transaction. Par conséquent, la valeur obtenue lors de l'appel à `account1.getSolde()` est incorrecte car la valeur a été modifiée par le *rollback* tardif.

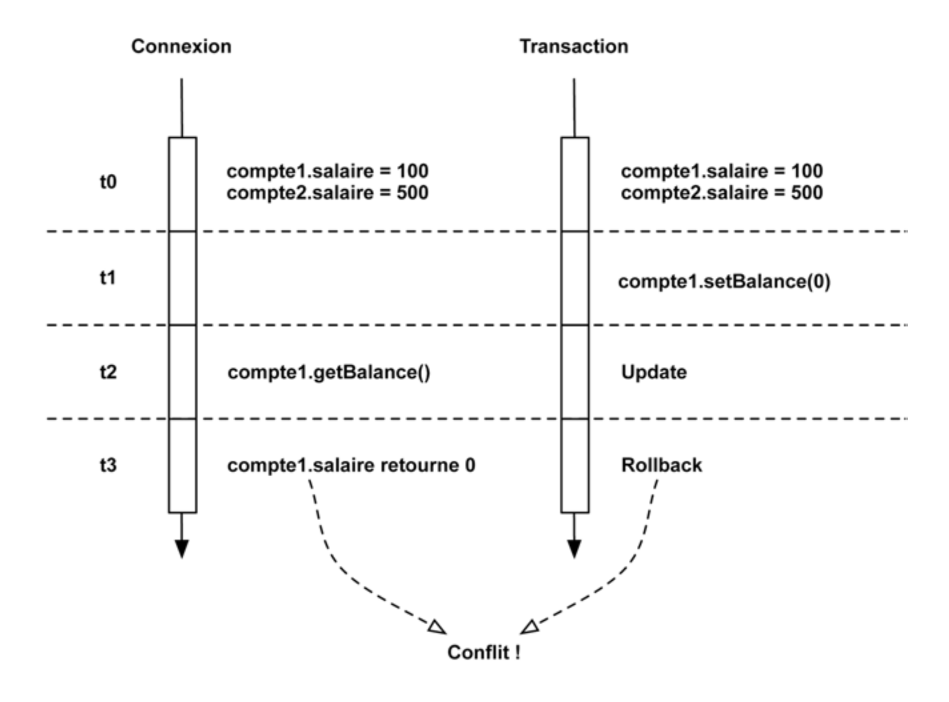


Figure 9.3 — Exemple « lecture sale »

- « *non repeatable reads* » : une « lecture non répétable » se produit lorsqu'une transaction lit une même donnée deux fois, et qu'une autre connexion modifie la donnée entre les deux lectures (fig. 9.4).

  Par exemple, nous voulons périodiquement rechercher les comptes à découvert et les approvisionner avec un autre compte (ayant un solde positif) du même client. Ceci s'effectue en deux étapes distinctes. Cependant, si une autre connexion modifie un des comptes entre ces étapes, celui-ci peut alors se retrouver dans la liste des comptes à découvert et dans celle des comptes disponibles pour le réapprovisionnement. Par conséquent, les listes deviennent obsolètes. Cette erreur est peu commune, et se produit pour les transactions lisant plusieurs fois le même objet.

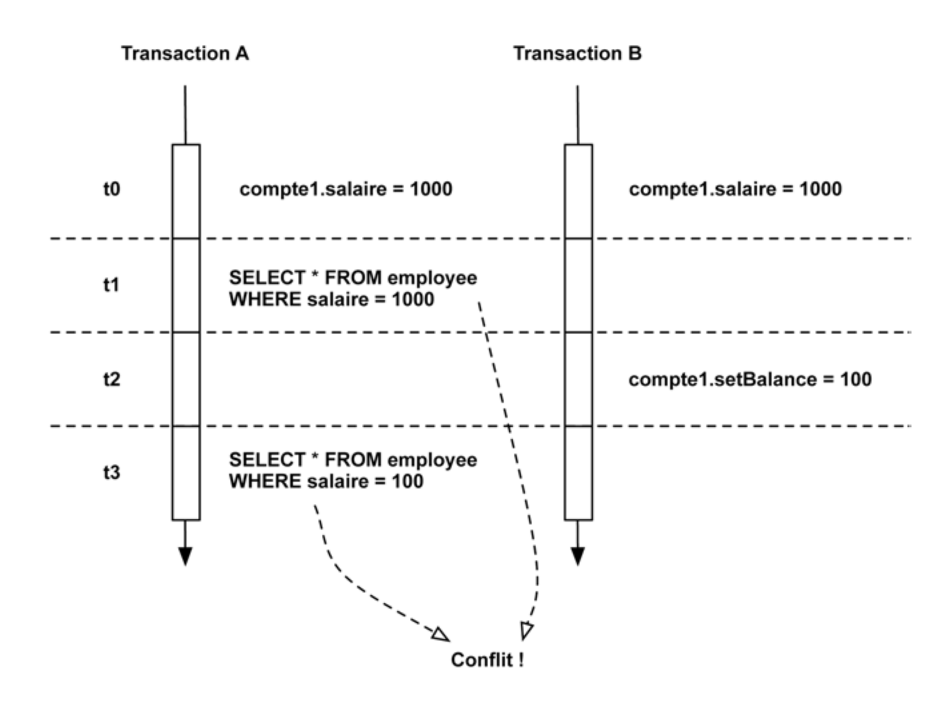


Figure 9.4 — Exemple « lecture non répétable »

- « phantom reads » : une « lecture fantôme » se produit lorsqu'une transaction effectue une recherche retournant un nombre variable d'objets, mais qu'une autre connexion ajoute un objet correspondant aux critères de cette recherche. La recherche effectuée ne sera alors plus en cohérence avec la source de données. Cette erreur est bien plus rare que les précédentes (fig. 9.5).

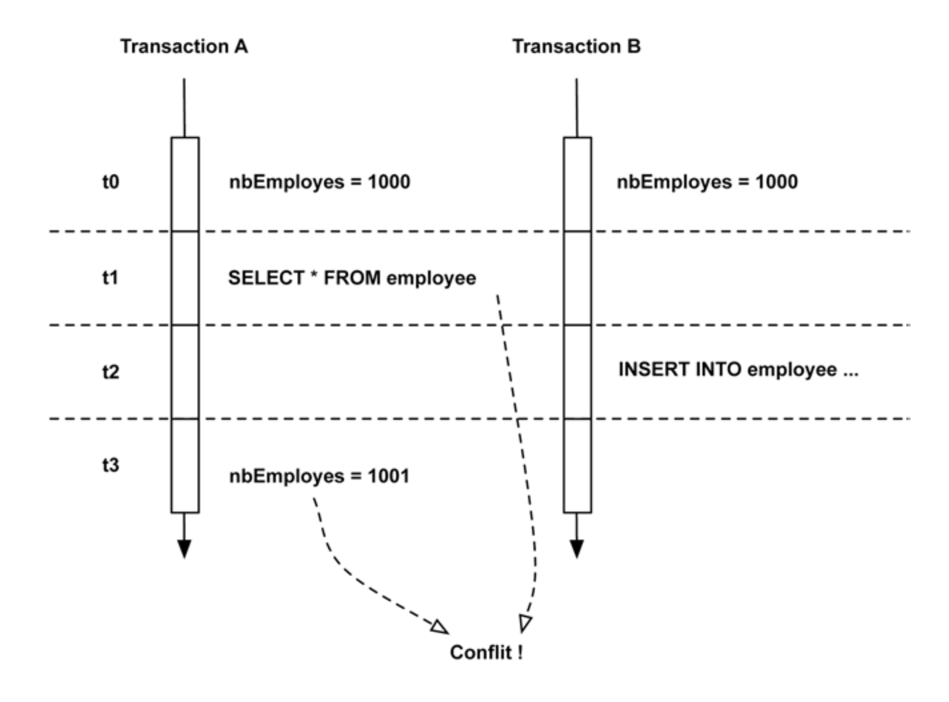


Figure 9.5 — Exemple « lecture fantôme »

Vous devez bien évaluer le niveau d'isolation nécessaire aux transactions d'une application. En effet, un degré fort d'isolation nuit aux performances d'exécution de vos opérations.

Vous avez à votre disposition, différents niveaux permettant d'appliquer une isolation plus ou moins complète sur vos transactions. Ceux-ci permettent de définir la façon dont les transactions sont isolées les unes des autres par rapport à la lecture des données.

Voici une liste des niveaux d'isolation que l'on retrouve le plus souvent au niveau des SGBD ou des systèmes d'information d'entreprise (EIS) :

- TRANSACTION_READ_UNCOMMITED : c'est le niveau le plus dangereux, mais le plus rapide à l'exécution. Une transaction peut lire des données non validées qui ont pu être modifiées par les transactions concurrentes. Il n'est pas conseillé d'utiliser ce niveau dans la plupart des contextes transactionnels.
- TRANSACTION_READ_COMMITED : c'est l'inverse du niveau précédent. Une transaction ne peut pas lire les changements non validés des transactions concurrentes. Ce mode est généralement utilisé pour les opérations fréquentes de lecture de données.

- TRANSACTION_REPEATABLE_READ : ce niveau assure que la lecture une même donnée à différents moments de la transaction procure le même résultat. Les données retournées sont les mêmes tout au long de la transaction, même si elles sont modifiées par une autre connexion.
- TRANSACTION_SERIALIZABLE : c'est le niveau le plus complet. La transaction a l'accès exclusif sur les données en positionnant un verrou (*lock*). Ce niveau est cependant à utiliser avec parcimonie. Même s'il assure une parfaite isolation transactionnelle, c'est un grand consommateur de performances. En effet, il assure que les données lues ont été validées, qu'elles restent identiques tout au long de la transaction et qu'aucune donnée n'a été ajoutée entre-temps.

Attention : même si ce choix semble être le plus adéquat, l'utilisation trop importante du niveau « TRANSACTION_SERIALIZABLE » peut conduire à l'immobilisation de la base de données.

Le tableau 9.1 récapitule les différents niveaux d'isolation avec la répercussion des erreurs possibles.

Tableau 9.1 — Récapitulatif des différents niveaux d'isolation

Niveau d'isolation	Lecture sale	Lecture répétable	Lecture fantôme
TRANSACTION_READ_UNCOMMITED	Possible	Possible	Possible
TRANSACTION_READ_COMMITED	Aucune	Possible	Possible
TRANSACTION_REPEATABLE_READ	Aucune	Aucune	Possible
TRANSACTION_SERIALIZABLE	Aucune	Aucune	Aucune

Ces niveaux d'isolation sont paramétrés par le gestionnaire de ressources grâce au conteneur (EJB par exemple) et plus généralement *via* les API (JDBC, JTA…).

Il faut tout de même, garder à l'esprit que le principe d'isolation n'est qu'une solution aux problèmes de concurrence d'accès qui peuvent se produire au sein d'une même table d'une base de données. Toutefois, il n'affecte pas les opérations internes d'une transaction mais seulement les transactions simultanées.

Lorsque vous utilisez des transactions globales, vous devez définir le niveau d'isolation pour chaque ressource utilisée.

> **Conseil :** essayez de n'avoir qu'un seul niveau d'isolation sur l'ensemble d'une transaction qui utilise plusieurs gestionnaires de ressources (transaction globale).

9.2 LES TRANSACTIONS DANS JAVA EE

Une transaction est définie par un début et une fin au sein d'un programme. En environnement Java EE, le paramétrage du gestionnaire de transactions peut se faire de trois manières :

- Être explicitement précisé dans le code.
- Être défini dans le descripteur de déploiement.
- Être géré implicitement par le conteneur.

Une transaction est démarquée, généralement, par l'appel de la méthode `begin()` et par l'appel des méthodes `commit()` ou `rollback()` pour mettre fin à cette dernière.

Dans le cas d'une transaction locale, le gestionnaire de ressources se charge de gérer implicitement les transactions. Cependant, vous devez définir les ordres de début et de fin de transaction *via* l'interface du gestionnaire de ressources. La situation la plus courante est l'accès direct à un SGBD avec l'API JDBC (*Java DataBase Connectivity*). Les ordres SQL envoyés sont validés par l'appel de l'instruction *commit* ou annulés *via rollback*. Toutefois, le développeur a la possibilité de définir le point d'entrée de la transaction (*via* `beginTransaction()` dans JDBC) et sa fin (*via* `commit()` ou `rollback()`).

Le cas des transactions globales est plus complexe. L'utilisation de l'API JTA (*Java Transaction API*) / JTS (*Java Transaction Service*) s'impose afin d'accéder au gestionnaire de ressources externes. JTS permet la démarcation, la connexion aux ressources, la synchronisation et la propagation du contexte transactionnel. Les EJB utilisent cette API et vous donnent le choix dans la manière de gérer ces transactions :

- Implicitement en utilisant le conteneur EJB et en spécifiant que les transactions du Bean sont gérées par le conteneur.
- Explicitement *via* l'interface `javax.transaction.UserTransaction`.

Le bénéfice de JTA est de pouvoir se connecter, au sein d'une même transaction, à différents composants d'accès aux données. Le développeur a une interface unique et n'a pas à gérer les différences d'implémentation.

9.2.1 Les transactions gérées par le conteneur EJB

Le conteneur EJB propose un service sur lequel toute la gestion des transactions peut être définie au sein des EJB. Toutefois, même si cette solution retire tout paramétrage spécifique dans vos EJB, vous devez en connaître son fonctionnement afin d'en tirer pleinement parti.

L'utilisation des transactions de type « *Container Managed* » (CMT, gérées par le conteneur) implique que la gestion des démarcations est à la charge du conteneur. Ceci permet de ne pas écrire le code transactionnel dans les EJB, mais de simplement faire le paramétrage des transactions dans le descripteur de déploiement. Le conteneur gère alors automatiquement la démarcation transactionnelle, suivant le paramétrage effectué.

Voici les différents types de transactions disponibles :

- « *Required* » (requis) : la plupart du temps utilisé pour les premiers tests, c'est sans doute le meilleur choix pour une méthode EJB transactionnelle.

  Si la méthode appelante est incluse dans une transaction, le conteneur n'en crée pas de nouvelle, mais utilise la méthode existante (fig. 9.6A). Dans le cas contraire, il crée une nouvelle méthode pour l'appel de la méthode transactionnelle (fig. 9.6B).

  Si une erreur survient lors de l'exécution de la méthode, l'ensemble des opérations est annulé (*rollback*), de même pour la méthode appelante, si elle était dans la transaction.

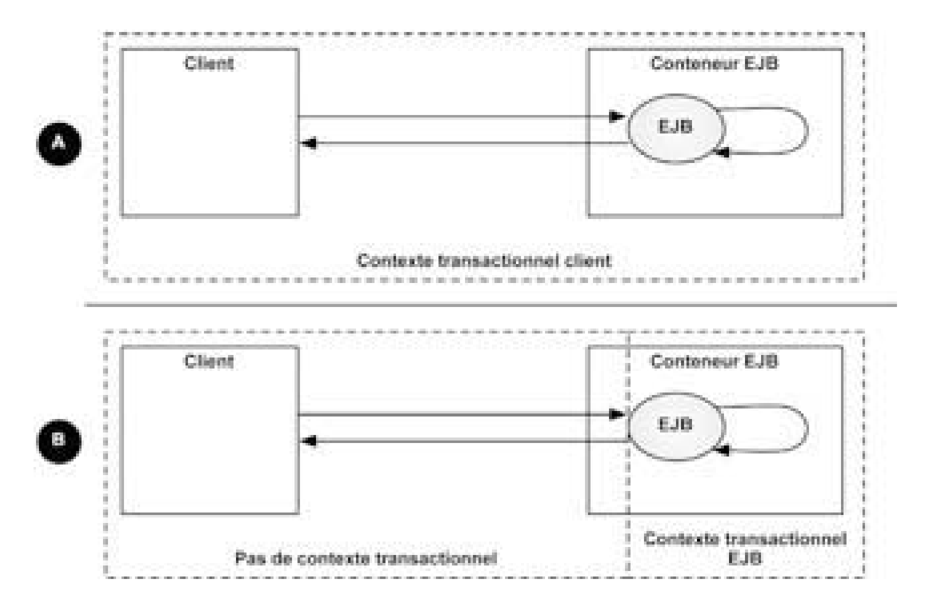


Figure 9.6 — Attribut « Required »

- « *RequiresNew* » (nouveau requis) : à utiliser lorsque vous devez être certain que la méthode soit annulée en cas de problème mais sans propager cette annulation à la méthode appelante.

  Si la méthode appelante s'exécute au sein d'une transaction, alors celle-ci est suspendue jusqu'à ce que la méthode appelée soit terminée (fig. 9.7).

  L'utilisation importante de cette configuration peut entraîner des pertes de performances. En effet, une nouvelle transaction est créée à chaque appel de la méthode.

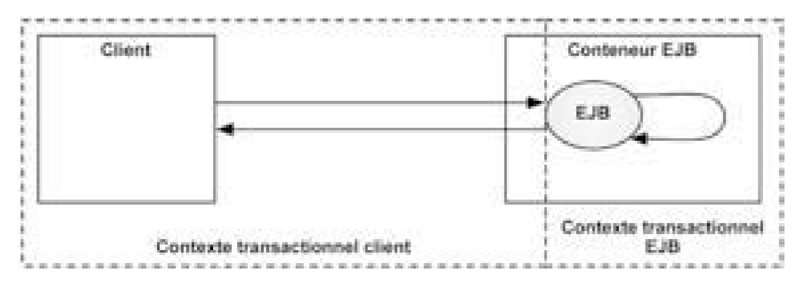


Figure 9.7 — Attribut « RequiresNew »

- « *Mandatory* » (obligatoire) : utilisé pour obliger une méthode à s'exécuter au sein d'un contexte transactionnel déjà existant.

  Si l'appelant est associé à une transaction, la méthode sera invoquée au sein de celle-ci (fig. 9.8A). Dans le cas contraire, une exception javax.transaction.TransactionRequiredException est lancée (fig. 9.8B).

  Cet attribut est utile lorsqu'un bean est appelé par d'autres Beans ou qu'il fait parti d'un ensemble important d'opérations.

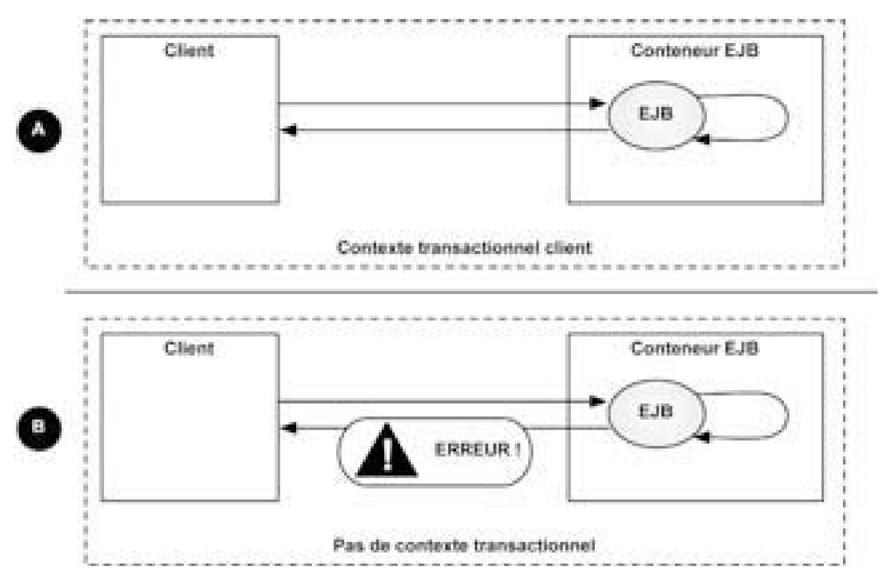


Figure 9.8 — Attribut « Mandatory »

- « *Supports* » (supporté) : la méthode est exécutée dans une transaction, si et seulement si l'appelant est associé à une transaction (fig. 9.9A). Dans l'autre cas, la méthode s'exécute sans contexte transactionnel (fig. 9.9B).

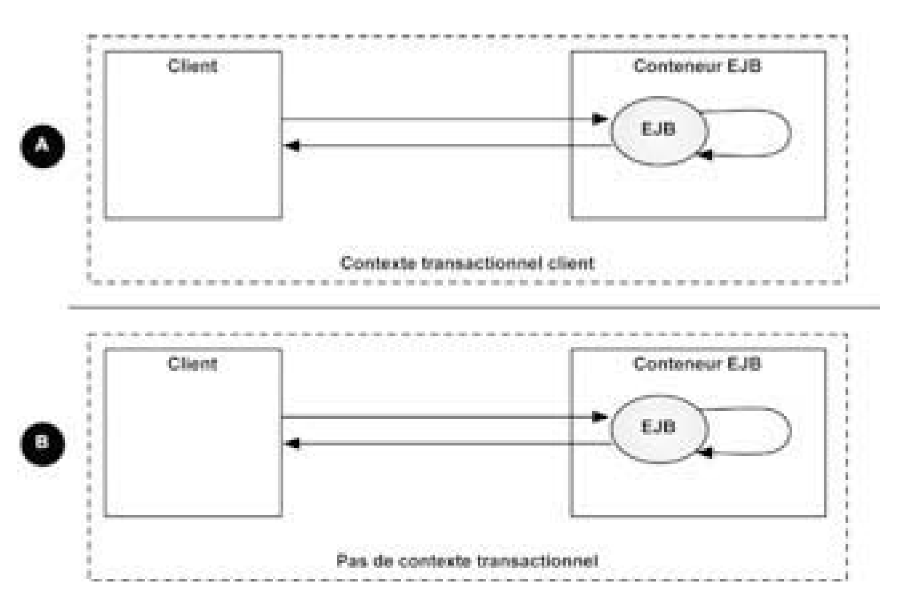


Figure 9.9 — Attribut « Supports »

- « *NotSupported* » (non supporté) : utilisé lorsque vous êtes sûr que votre méthode n'a pas besoin de contexte transactionnel (fig. 9.10) et que celle-ci n'effectue pas d'opérations critiques (c'est-à-dire que son utilisation vis-à-vis d'autres opérations simultanées n'a pas besoin d'être isolée).

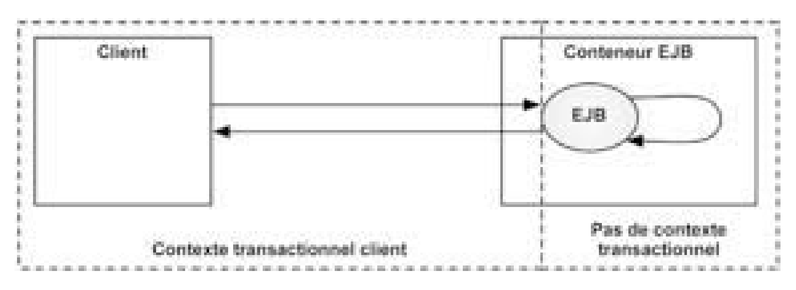


Figure 9.10 — Attribut « NotSupported »

- « *Never* » (jamais) : utilisé lorsque vous voulez être sûr que les clients n'utilisent pas de transactions lors de l'appel de la méthode.

  Interdit l'exécution de la méthode au sein d'un contexte transactionnel. La méthode ne peut être appelée si et seulement si aucun contexte transactionnel n'existe (fig. 9.11A).

  Si l'appelant est associé à une transaction, alors une erreur `javax.ejb.Exception` (local) ou `java.rmi.RemoteException` (distant) est lancée (fig. 9.11B).

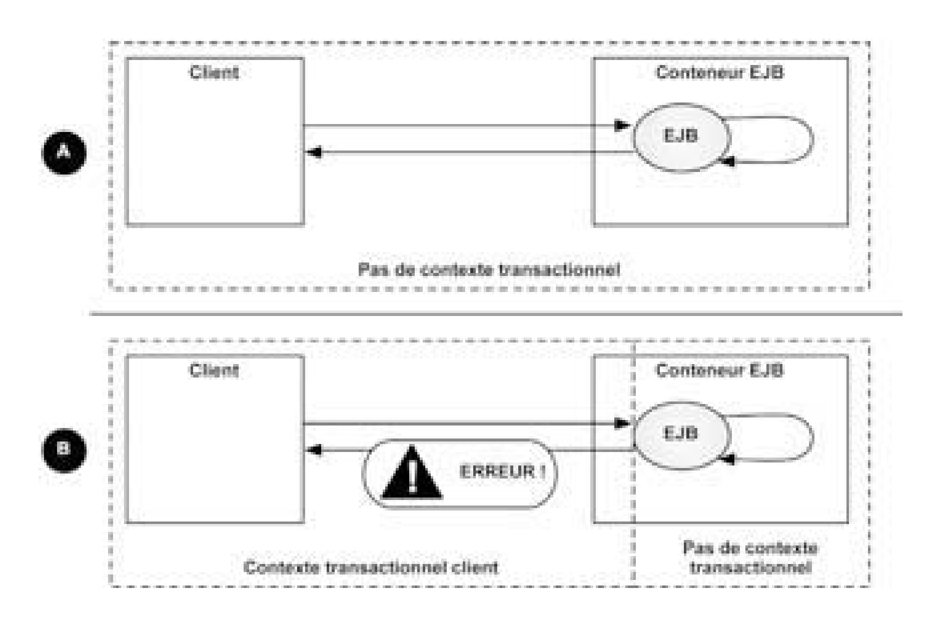


Figure 9.11 — Attribut « Never »

Le tableau 9.2 présente un récapitulatif des différents types de transaction.

Tableau 9.2 — Récapitulatif des différents types de transaction

Attributs transactionnels	Contexte transactionnel parent existant	Contexte transactionnel parent NON existant
REQUIRED	Exécution dans le contexte existant	Création d'un nouveau contexte
REQUIRESNEW	Création d'un nouveau contexte	Création d'un nouveau contexte
MANDATORY	Exécution dans le contexte existant	Levée d'exception : javax.Transaction.TransactionRequiredException
SUPPORTS	Exécution dans le contexte existant	Aucune transaction
NOT SUPPORTED	Hors contexte transactionnel existant	Hors contexte transactionnel
NEVER	Levée d'exception : javax.ejb.Exception ou java.rmi.RemoteExption	Hors contexte transactionnel

Notez que ces attributs ne peuvent pas être appliqués à tous les types de Bean, comme il est précisé dans le tableau 9.3.

Tableau 9.3 — Récapitulatif des cas d'utilisation des différents types de transaction

Attributs transactionnels	Stateless	Stateful	Entity	Message Driven
REQUIRED	X	X	X	X
REQUIRESNEW	X	X	X	
MANDATORY	X	X	X	
SUPPORTS	X			
NOT SUPPORTED	X			
NEVER	X			X

Une transaction est validée (*commit*) dès que l'exécution de la méthode, ayant initialisée la transaction, se termine et que la réponse est retournée au client. A contrario, la transaction est annulée (*rollback*) si une exception est levée. Cependant, suivant le type de l'exception lancée, la gestion de la transaction se fera différemment. Le tableau 9.4 récapitule cette gestion.

> **Remarque :** une exception applicative correspond aux cas d'erreurs fonctionnelles. Toutes les exceptions doivent apparaître dans le prototype de la méthode (mot-clé `throws`). Une exception système (autres exceptions) correspond aux erreurs de bas niveau telles les problèmes de connexion à une base de données, à un service distant… Elles couvrent l'ensemble des erreurs inattendues.

Le principe des transactions ne change guère entre la spécification EJB 2 et EJB 3. Néanmoins, le changement important intervient dans la définition et la déclaration des attributs transactionnels. Cette définition a été simplifiée en EJB 3 par l'utilisation des attributs.

Tableau 9.4 — Récapitulatif des liaisons entre transactions et exceptions

Condition d'appel (t=0)	Exception levée (t=1)	Action du conteneur (t=2)	Vue de l'appelant (t=3)
Méthode appelée dans le contexte transactionnel de l'appelant (Required, Mandatory et Supports)	Exception applicative	Relance l'exception	Reçoit l'exception Applicative Peut continuer la transaction sauf si l'instance appelle la méthode ***setRollbackOnly()***
	Autres exceptions ou erreurs	Transaction marquée Rollback : TransactionRolledback-Exception ou TransactionRolledback-LocalException	Reçoit TransactionRolledback-Exception ou TransactionRolledback-LocalException Impossible de continuer la transaction
Méthode appelée dans le contexte transactionnel initié au début de la méthode (Required, RequiresNew)	Exception applicative	Si l'instance appelle la méthode setRollbackOnly() alors : Relance l'exception applicative + Transaction marquée Rollback Sinon la méthode tente de réaliser le commit puis relance l'exception applicative	Reçoit l'exception applicative La transaction éventuelle de l'appelant peut se poursuivre
	Autres exceptions ou erreurs	Transaction initiée marquée RollbackRemoteException ou EJBException	Reçoit RemoteException ou EJBException La transaction éventuelle de l'appelant peut se poursuivre
Méthode appelée sans contexte transactionnel (NotSupported, Never, Supports)	Exception applicative	Relance l'exception Applicative	Reçoit l'exception Applicative La transaction éventuelle de l'appelant peut se poursuivre.
	Autres exceptions ou erreurs	RemoteException ou EJBException	Reçoit RemoteException ou EJBException La transaction éventuelle de l'appelant peut se poursuivre

Avec EJB 2

Avec EJB 2, la démarcation des transactions se fait automatiquement par le conteneur, vous devez cependant préciser et paramétrer les méthodes transactionnelles *via* le descripteur de déploiement.

Voici une partie d'un descripteur de déploiement définissant un contexte transactionnel par défaut (ensemble du Bean) et pour une méthode donnée :

```
...
<container-transaction >
```

```
    <method >
      <ejb-name>RegisterAccount</ejb-name>
       <method-name>*</method-name>
     </method>
     <trans-attribute>Required</trans-attribute>
  </container-transaction>

 <container-transaction >
    <method >
      <ejb-name>RegisterAccount</ejb-name>
      <method-intf>Local</method-intf>
      <method-name>createAccount</method-name>
      <method-params>
         <method-param>java.lang.Integer</method-param>
         <method-param>java.lang.String</method-param>
         <method-param>java.lang.String</method-param>
         <method-param>java.lang.String</method-param>
         <method-param>java.lang.String</method-param>
      </method-params>
    </method>
    <trans-attribute>Supports</trans-attribute>
 </container-transaction>
 ...
```

La première définition de transaction déclare un paramétrage par défaut pour l'ensemble des méthodes. Ce paramétrage porte sur l'EJB ayant le nom `RegisterAccount` et le type de transaction est `Required`.

La seconde définition déclare la méthode locale `createAccount` ayant cinq paramètres (un `Integer` et 4 `String`) avec une transaction de type `Supports`.

Par conséquent, la définition par défaut (la première) est alors sur-définie et remplacée par la plus précise (la seconde).

Avec EJB 3

La configuration des transactions en EJB 3 est vraiment facilitée. Il suffit, désormais, d'annoter simplement la méthode transactionnelle avec `@TransactionAttribute`. Cette annotation prend en paramètre le type de transaction à assigner *via* l'énumération `TransactionAttributeType`. Celle-ci encapsule les types de transactions : MANDATORY, NEVER, NOT_SUPPORTED, REQUIRED, REQUIRES_NEW, SUPPORTS.

Voici un exemple de code simple :

```
@Stateless public class MySessionBean implements MySession {
//...
  @TransactionAttribute(TransactionAttributeType.SUPPORTS)
  public void someMethod(...) {
    java.sql.Connection con1;
    java.sql.Connection con2;
    java.sql.Statement stmt1;
    java.sql.Statement stmt2;
    // Récuperation de deux connexions
    connect1 = ...;
```

```
    connect2 = ...;
    stmt1 = connect1.createStatement();
    stmt2 = connect2.createStatement();
    //
    // La modification des données sur les deux bases de données. Le conteneur
    // empile automatiquement connect1 et connect2 avec la transaction gérée
    // par le conteneur.
    //
    stmt1.executeQuery(...);
    stmt1.executeUpdate(...);
    stmt2.executeQuery(...);
    stmt2.executeUpdate(...);
    stmt1.executeUpdate(...);
    stmt2.executeUpdate(...);
    // déconnexions
    connect1.close();
    connect2.close();
  }
  //...
}
```

Les connexions et les opérations de la méthode `someMethod()` sont empilées dans le contexte transactionnel automatiquement. Vous n'avez pas à définir le début et la fin de la transaction. La démarcation est établie par le conteneur grâce aux annotations transactionnelles.

Il est également possible d'utiliser cette annotation au niveau de la classe du composant. À partir de là, le type de transaction attribué est alors appliqué à l'ensemble des méthodes non marquées.

```
@Stateless
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public class RichClientServiceBean implements RichClientService {

  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void updateProfile(User user) {
    ...
  }

  // TransactionType.SUPPORTS
  public void disconnect() { ... }

}
```

Dans l'exemple précédent, la méthode `updateProfile()` surdéfinit le type de transaction appliquée. Si aucune surdéfinition n'est déclarée, alors la méthode utilise la définition de la classe (cf. méthode `disconnect()`).

Si aucun type de transaction n'est défini *via* `@TransactionAttribute`, alors le type `TransactionAttributeType.REQUIRED` est appliqué par défaut. En effet, dans la majorité des cas, les méthodes des EJB sont transactionnelles surtout si elles utilisent l'EntityManager !

9.2.2 Les transactions gérées par le Bean

La gestion des transactions par le Bean doit être utilisée seulement lorsque le mode de gestion par le conteneur ne peut être appliqué. En effet, nous vous conseillons d'utiliser au maximum les services du conteneur, en se basant sur la spécification et donc d'être sûr de sa portabilité. Cependant, cette méthode offre une gestion plus précise de vos transactions. Contrairement au mode « container managed » qui démarque la transaction au niveau d'une méthode complète, la gestion par le bean permet une démarcation beaucoup plus fine (au sein même de la méthode).

La transaction peut être initialisée et gérée directement dans vos Beans (Session et Message Driven seulement). Le principe consiste simplement à récupérer un objet de type `javax.transaction.UserTransaction` et d'en appeler la méthode `begin()` pour initialiser la transaction. La fin de celle-ci se fera par l'appel des méthodes `commit()` ou `rollback()` pour valider ou annuler cette transaction.

Voici une description des méthodes de l'interface `UserTransaction` :

- `void begin()` : crée une nouvelle transaction et l'assigne au processus courant.
- `void commit()` : valide la transaction assignée au processus courant.
- `int getStatus()` : retourne le statut de la transaction courante.
- `void rollback()` : annule la transaction courante.
- `void setRollbackOnly()` : modifie la transaction courante afin d'en signifier l'annulation (*rollback*). La transaction ne pourra alors plus être validée.
- `void setTransactionTimeout(int seconds)` : modifie la valeur du temps maximum d'exécution de la transaction courante (temps depuis l'appel de la méthode `begin()`).

Le principe de gestion de la transaction est le même pour les EJB 2 et les EJB 3. Seule la façon de récupérer l'objet de type `UserTransaction` diffère.

Avec EJB 2

Avec les EJB 2, il est nécessaire d'utiliser l'objet `SessionContext`, associé à l'instance de l'EJB, afin de récupérer l'objet `UserTransaction`, comme le montre l'exemple suivant :

```
…
SessionContext sessionContext;

// méthode de l'interface SessionBean utilisée
// par le conteneur pour « setter » l'objet SessionContext
public void setSessionContext(SessionContext session) throws EJBException,
RemoteException {
  this.sessionContext = session;
}

public void transfer(Account sourceAccount, Account targetAccount, Integer
amount) {
```

```
…
session.getUserTransaction().begin();
// appels de méthodes dans le contexte transactionnel
…
// lorsqu'il y a un problème, on annule la transaction
session.getUserTransaction().rollback();
…
// tout s'est bien passé, on valide les opérations
session.getUserTransaction().commit();
…
}
```

La méthode `setSessionContext(SessionContext session)` permet de récupérer l'objet `SessionContext` transmis par le conteneur. Il suffit, alors, d'appeler la méthode `getUserTransaction()` qui retourne un objet `UserTransaction`.

Il faudra également spécifier que les transactions au sein de votre Bean sont de type : « Bean Managed » (gérée par le Bean). Cela devra être précisé dans le descripteur de déploiement, comme dans l'exemple suivant :

```
<session>
    <description><![CDATA[]]></description>
    <ejb-name>RichClientService</ejb-name>

    …
    <session-type>Stateless</session-type>
    <transaction-type>Bean</transaction-type>
 </session>
```

Avec EJB 3

La spécification EJB 3 simplifie l'accès aux ressources externes. Pour récupérer l'objet `UserTransaction`, il vous suffit d'utiliser l'annotation `@Resource` sur la propriété du Bean (variable d'instance). Celle-ci sera alors automatiquement initialisée par le conteneur lors de l'accès au Bean.

Tout comme pour les EJB 2, il sera nécessaire de préciser que les transactions sont de type « Bean Managed ». Cependant, c'est grâce à l'annotation `@TransactionManagement` sur la classe du Bean que ce paramétrage se fera avec EJB 3.

Voici un exemple de code :

```
@Stateful
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
  @Resource javax.Transaction.UserTransaction ut;
  @Resource javax.sql.DataSource database1;
  public void method1(...) {
    // démarre une transaction
    ut.begin();
  }
  public void method2(...) {
    java.sql.Connection con;
    java.sql.Statement stmt;
    // ouvre une connexion
    con = database1.getConnection();
```

```
    // modifications des données
    stmt = con.createStatement();
    stmt.executeUpdate(...);
    stmt.executeUpdate(...);
    // ferme la connexion
    stmt.close();
    con.close();
  }
  public void method3(...) {
    // commit de la transaction
    ut.commit();
  }
  ...
}
```

Vous pouvez remarquer que seule l'initialisation de l'objet `UserTransaction` change entre EJB 2 et EJB 3.

9.3 SCÉNARIOS D'UTILISATION

Afin de mieux comprendre à quel moment utiliser les transactions, voici quelques exemples ou cas pratiques illustrant des scénarios possibles dans le développement d'application.

9.3.1 Modifications sur plusieurs bases de données

Vous pouvez être amenés à devoir travailler, au sein d'une même opération, avec plusieurs bases de données. C'est souvent le cas dans les applications bancaires. En effet, pour des raisons de performances, de sécurité ou de facilité de maintenance, les bases de données des comptes bancaires et/ou des mouvements peuvent être hébergés sur différents serveurs.

Lorsqu'une opération de virement doit être faite, il faut alors modifier différents comptes et enregistrer le mouvement. Une transaction globale doit alors être mise en place pour assurer l'intégralité des opérations et des données (fig. 9.12).

Dans cet exemple, le client appelle l'EJB n1. Celui-ci modifie des données sur deux bases de données A et B (configurées sur deux serveurs différents au sein du serveur d'applications) puis appelle l'EJB n2. Celui-ci met à jour une troisième base de données C.

Le serveur EJB a la responsabilité de valider ou d'annuler les opérations. Le programmeur, quant à lui, n'a pas à se focaliser sur le code de gestion de la transaction. En outre, le serveur EJB empile l'ensemble des connexions aux bases de données dans la transaction et gère le cycle de vie de celle-ci.

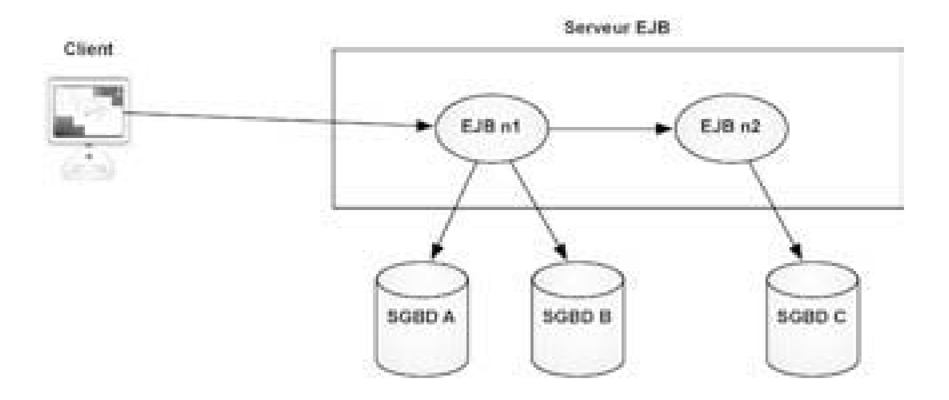


Figure 9.12 — Modification de plusieurs bases de données

9.3.2 Appel JMS et modifications sur plusieurs bases de données

De la même façon que dans le scénario précédent, il se peut qu'un système externe puisse interagir avec le serveur EJB *via* JMS (*Java Message Service*). Inversement un EJB peut envoyer un/des message(s) à un serveur JMS lors de modifications pour avertir un système externe.

La spécification EJB prend en compte ce cas de figure au sein d'une même transaction (fig. 9.13).

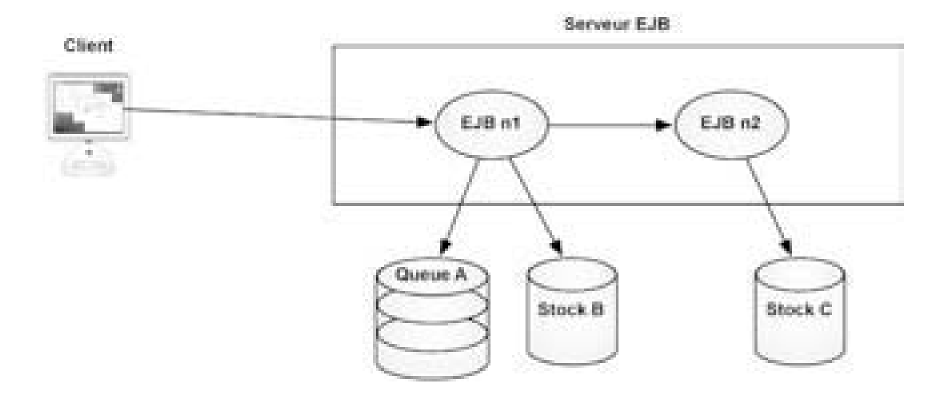


Figure 9.13 — Modifications de plusieurs bases de données et envoi d'un message JMS

Dans cet exemple, un client appelle l'EJB n1. Celui-ci envoie un message à la queue JMS et effectue les mêmes opérations que dans l'exemple précédent. Le serveur EJB empile l'ensemble des appels (JDBC/JMS) et valide ou annule l'ensemble.

9.3.3 Modifications avec plusieurs serveurs EJB

Le premier scénario peut tout à fait se constituer de plusieurs serveurs EJB, pour un souci de maintenance ou de performance, par exemple. La transaction sera alors propagée sur l'ensemble des serveurs EJB.

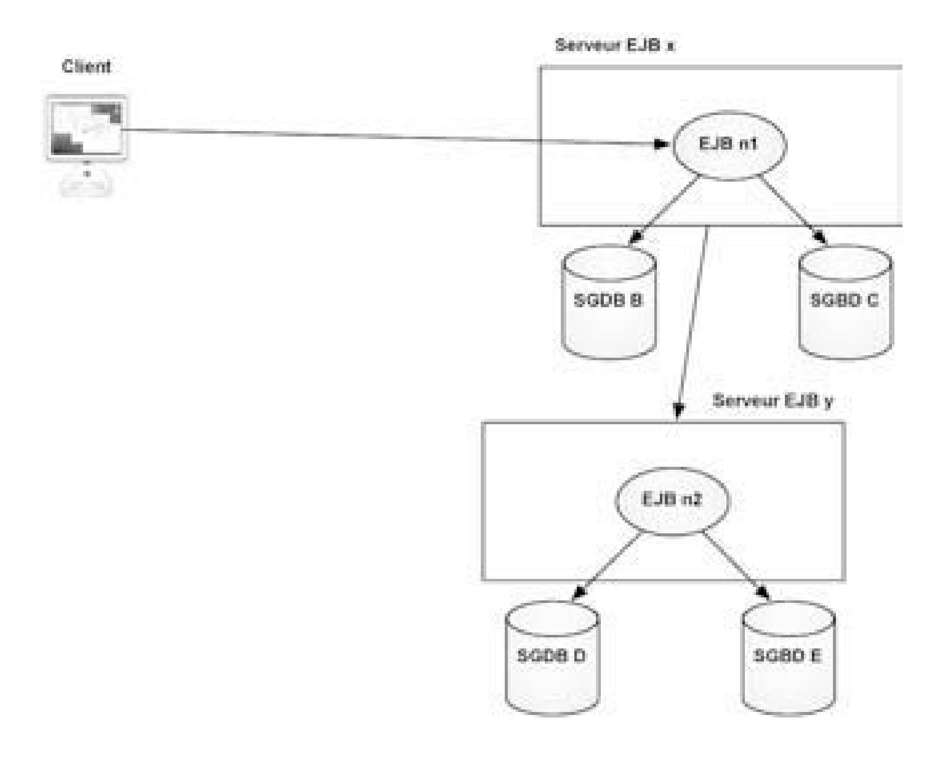


Figure 9.14 — Modifications de plusieurs bases de données avec plusieurs serveurs EJB

Dans l'exemple de la figure 9.14, un client appelle un EJB n1 sur le premier serveur EJB. Celui-ci met à jour les bases de données B et C. Ce serveur EJB appelle ensuite le second serveur EJB pour répliquer les modifications sur l'EJB n2. Ce dernier met à jour la base de données B.

Les deux serveurs coopèrent afin de propager le contexte transactionnel (cela est totalement transparent pour le développeur). Lorsque l'appel est terminé, les deux serveurs utilisent un protocole à deux phases de validation pour assurer l'atomicité des modifications sur les bases de données.

En résumé

Dans la même lignée que les simplifications des EJB, les transactions sont désormais paramétrables directement avec les annotations. De même, leur utilisation est plus intuitive et les possibilités d'optimisations des applications plus vastes.

10

Les outils indispensables

Objectif

La conception d'EJB est loin de s'arrêter à l'utilisation de la machine virtuelle et du compilateur. En effet, l'étape indispensable que nous allons voir consiste à choisir ses outils, tels que le serveur d'applications, l'éditeur et les moteurs de scripts d'automatisation, parmi ceux proposés par différents fournisseurs.

Nous conseillons de tester les outils que nous présentons dans ce chapitre, afin de sélectionner ceux adaptés à vos besoins et qui vous correspondront le mieux.

10.1 INTRODUCTION

Le serveur d'applications est à la base d'une application Java EE. Il fournit l'ensemble des dispositifs permettant de créer, tester et de mettre en production les services que l'on souhaite mettre en place avec les applications web, EJB…

Ils correspondent à la spécification fournie par un groupe d'experts composé de membres d'entreprises comme Sun, IBM, Bea, JBoss… Ces compagnies participent activement au monde de Java EE et proposent pour la plupart une implémentation particulière. À l'écriture de cet ouvrage, la majorité d'entre eux entame une phase de transition qui devrait durer jusqu'au début 2007, et des produits comme Sun Java System Application Server ou JBoss sont déjà utilisables.

De nombreux efforts ont été faits depuis les dernières spécifications afin de les uniformiser et de les rendre compatibles entre eux. Ainsi, tout ce que nous avons étudié dans les chapitres précédents doit fonctionner sur l'ensemble des implémen-

tations EJB 3. La différence entre eux provient de la facilité d'administration, du support offert par le constructeur, du prix…

Nous allons comparer les deux serveurs les plus populaires de l'Open Source supportant les EJB 3 et plus largement Java EE 5 :

- Sun Java System Application Server est l'implémentation de référence de Sun. De manière générale, on l'utilise plus facilement avec l'IDE NetBeans.
- JBoss, est le serveur proposé par une communauté de programmeurs, récemment racheté par Red-Hat.

Nous ne parlerons pas ici d'« Oracle Application Server », ni de « WebSphere », qui ne proposaient pas de support complet des EJB 3 lors de la rédaction.

10.2 LES SOLUTIONS DE SUN MICROSYSTEMS

10.2.1 Sun Java System Application Server

Historique

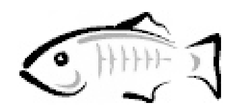


Figure 10.1 — Logo GlassFish

Sun Java System Application Server (SJSAS), proposé par Sun Microsystems, est à sa neuvième version. Il offre une implémentation de référence de Java EE jusqu'à la version 5.

Prénommé *GlassFish*, son grand avantage est d'avoir pu créer une communauté soutenant le projet. Leur site www.glassfish.org contient de nombreuses informations sur le fonctionnement du serveur, proposant également de rejoindre la communauté.

SJSAS est fourni avec *Sun Java System Message Queue*, un serveur JMS autonome, et *Sun Java System Web Server*, un serveur web.

> **Remarque :** la licence CDDL, appliquée à ce serveur, a été proposée par Sun Microsystems et est appliquée à certains de ses produits comme Solaris 10. Elle permet la modification du code source ainsi que sa réutilisation, s'inspirant largement de la licence de la communauté Mozilla. Cependant, elle n'est pas compatible avec la GPL.

Pré-requis

SJSAS supporte les systèmes d'exploitation suivants : Solaris 9 et 10 (SPARC Platform Edition, x86 Platform Edition), 64bit Solaris 10 (SPARC, x86), Microsoft

Windows 2000® Advanced Server, Microsoft Windows Server 2003, Microsoft Windows XP®, Red Hat Enterprise Linux 3.0 et 4.0, et Mac OS 10.4.

Au niveau *hardware*, le minimum matériel requis est 256 Mo de mémoire vive ainsi qu'un espace disque de 250 Mo.

Le support des bases de données suivantes est inclus : Oracle 8, 9 et 10, Microsoft SQL Server 2000, MSSQL 5, Sybase 12.5, IBM DB2 8.1, Derby 10.1.1 Embedded, PostGreSQL et MySQL.

SJSAS contient des librairies tierces comme TopLink, le moteur de persistance objet/relationnel d'Oracle. Le restant a été écrit, soit par Sun, soit par la communauté. Il supporte bien évidemment toutes les fonctionnalités de Java EE 5 : JNDI, JTA, JMS 1.1, JSP, EJB 3.0, WebServices…

SJSAS, même étant la référence d'implémentation de Java EE 5, reste compatible Java EE 1.4.

L'utilisation

Afin de tester le logiciel, nous récupérons l'installateur à l'adresse suivante : http://java.sun.com/javaee/downloads/index.jsp

Plusieurs solutions packagées sont disponibles. Prenez « Java EE 5 SDK », puis sélectionner la version destinée à votre système d'exploitation.

Une fois l'installateur téléchargé, il suffit de l'exécuter et vous pourrez ensuite démarrer le serveur. Le démarrage se fait soit par des raccourcis mis à disposition sur le bureau, ou par ligne de commande.

Dans le premier cas, sous Windows®, le menu « Démarrer » est plutôt clair (fig. 10.2).

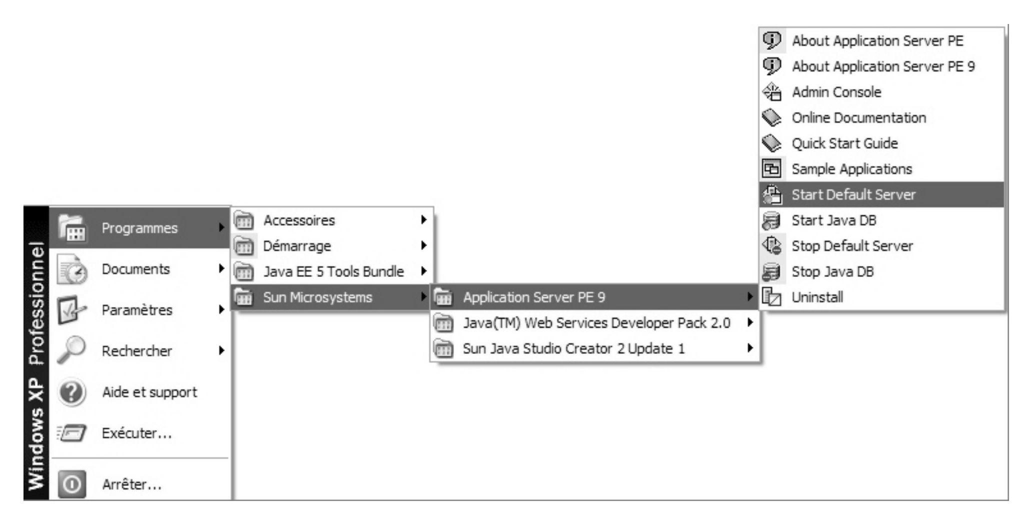


Figure 10.2 — Sun Application Server à partir du menu « Démarrer »

L'outil d'administration par ligne de commande est assez complet, mais nous ne citerons que la commande suivante permettant de démarrer le serveur :

- sous Windows® :

```
– cd <Répertoire de SJAS>\bin
– asadmin.bat start-domain
```

- sous Linux ou Solaris :

```
– cd <Répertoire de SJAS>/bin
– ./asadmin.sh start-domain
```

- sous Mac OS :

```
– cd <Répertoire de SJAS>/bin
– ./asadmin start-domain
```

Lors du démarrage, les informations suivantes sont affichées à l'écran :

```
Starting Domain domain1, please wait.
Log redirected to /Users/jbr/Applications/SUNWappserver/domains/domain1/logs/
server.log.Domain
domain1 is ready to receive client requests.
Additional services are being started in background.
Domain [domain1] is running [Sun Java System Application Server Platform Edition
9.0 (build b48)] with its configuration and logs at: [/Users/jbr/Applications/
SUNWappserver/domains].
Admin Console is available at [http://localhost:4848].
Use the same port [4848] for "asadmin" commands.

User web applications are available at these URLs:
[http://localhost:8080 https://localhost:8181 ].
Following web-contexts are available:
[/web1 /asadmin /ZooApplication ].
Standard JMX Clients (like JConsole) can connect to JMXServiceURL:
[service:jmx:rmi:///jndi/rmi://mycomputer.local:8686/jmxrmi] for domain
management purposes.
Domain listens on at least following ports for connections:
[8080 8181 4848 3700 3820 3920 8686 ].
```

Ces dernières informations correspondent aux réglages par défaut sur une machine avec Mac OS X.

Une fois ces informations affichées dans la console, nous pouvons accéder à l'interface d'administration *via* l'adresse http://localhost:4848/asadmin/index.html, où nous devrons nous authentifier à partir du login/password spécifié lors de l'installation. Ensuite, la page de la figure 10.3 s'affichera.

Cette partie permet de configurer l'ensemble de l'application (fig. 10.3), comme, par exemple, les pools de connexion JDBC (fig. 10.4 et 10.5), les différentes applications déployées…

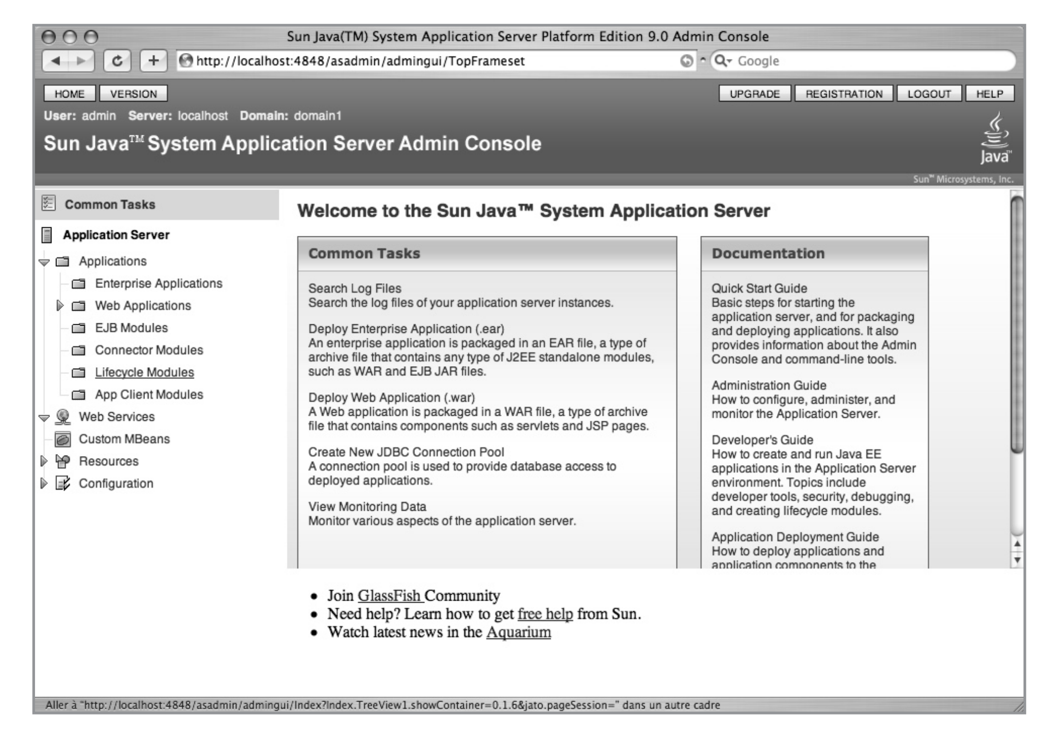


Figure 10.3 – Interface d'administration web de SJAS

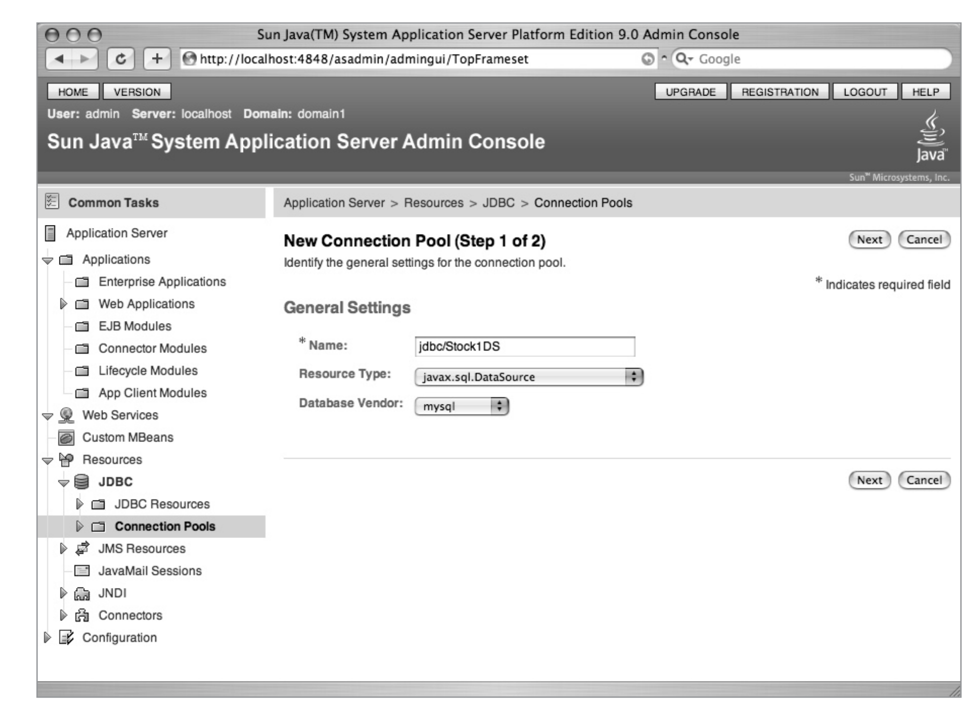
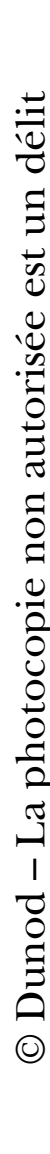


Figure 10.4 – Création d'un pool de connexion JDBC



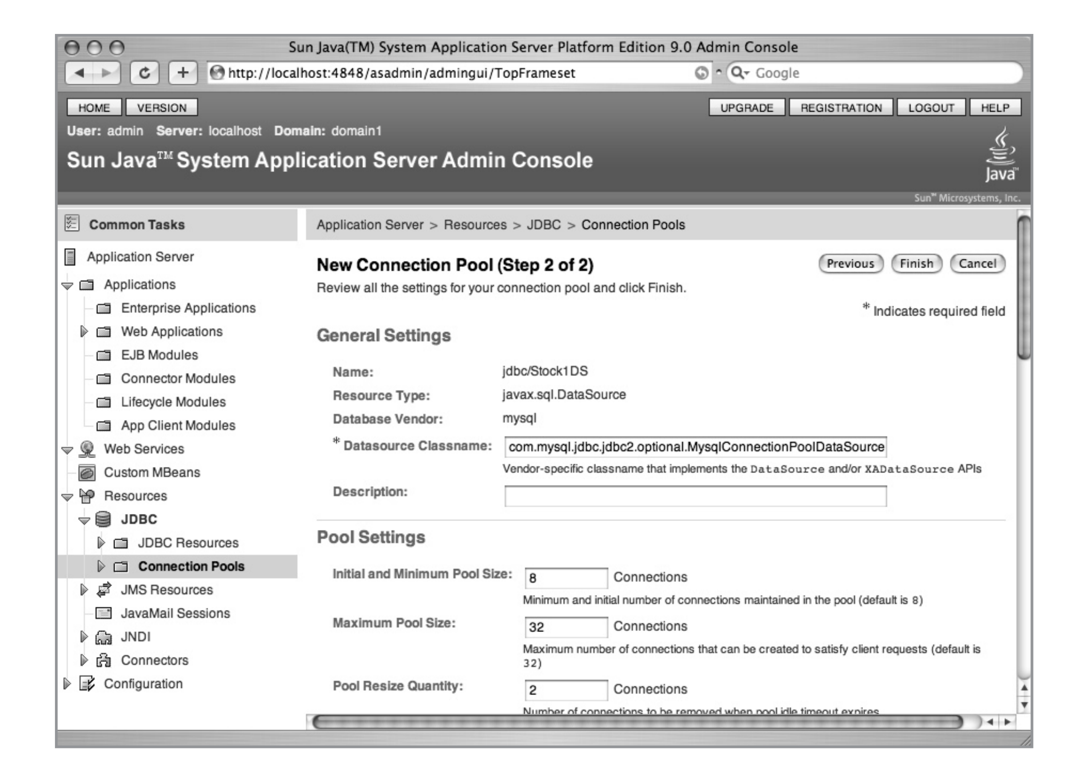


Figure 10.5 — Paramétrage du nouveau pool de connexion

Pour déployer vos services, vous avez deux options possibles :

- Utiliser cette interface graphique, puis selon le type d'application choisir « Entreprise Applications », « Web Applications », « EJB Modules »… un formulaire récupérera l'archive de l'application.
- Copier directement vos fichiers (war, jar ou ear) dans le répertoire « autodeploy » de votre serveur (par exemple : <répertoire d'installation de Sun AppServer 9>\domains\domain1\autodeploy). Dès qu'une archive est déposée dans ce répertoire, elle est automatiquement déployée.

Une dernière solution consiste à automatiser le déploiement par l'intermédiaire d'outils (IDE) qui vont se charger de réaliser cette tâche.

Nous n'entrerons pas d'avantage dans le fonctionnement interne de SJAS car cela serait superflu au vu de son intégration dans NetBeans.

L'environnement de développement est devenu un logiciel central dans le travail d'un développeur. Aujourd'hui de tels outils intègrent beaucoup de fonctions telles que la gestion du projet, l'édition graphique des fichiers de configuration, la supervision du serveur d'applications, le débogage, le déploiement…

Nous allons présenter NetBeans qui permet de se coupler très facilement à SJSAS. Nous nous intéresserons particulièrement aux fonctions natives ainsi qu'aux *plugins* adaptés à la création d'EJB 3.

10.2.2 NetBeans

Historique



Figure 10.6 — Logo NetBeans

NetBeans est inventé en 1996 par un Tchèque appelé Xelfi. Le but était d'écrire un IDE de type Delphi, pour Java et écrit en Java. En 1999, la version 3.0 sort, et l'éditeur est racheté par Sun. L'IDE sort sous le nom de « Sun Forte For Java Community Edition ». Celle-ci est gratuite, et la version professionnelle payante.

En juin 2000, Sun sort NetBeans, qui reprend son nom original, en Open Source.

L'IDE continue alors d'évoluer et, 4 ans plus tard, la version 4.0 sort avec le support des nouveautés du langage J2SE 5, une meilleure gestion des projets, une interface graphique améliorée et une utilisation accrue de Ant. La version 4.1, ajoute des modules J2EE 1.4 pour gérer les applications web ainsi que les EJB.

Enfin, en juin 2006, Sun Microsystems et la communauté de programmeurs de l'IDE lancent en fanfare la version 5.5, qui inclut un « profiler », un module de modélisation, des « Pack » permettant de supporter le développement pour Java Micro Edition… et bien évidemment Java EE 5.

Dans la politique de Sun visant à ouvrir et offrir ses logiciels, NetBeans bénéficie de plus en plus de soutien. Il est ainsi devenu l'outil de développement utilisé par les développeurs de projets comme Glassfish. Le but étant de le tester d'avantage et de le rendre de plus en plus performant.

Remarque : NetBeans est librement téléchargeable, pour la majorité des systèmes, sur le site http://www.netbeans.org.

Parmi les IDE de Sun Microsystems, Java Studio Creator 2 (SJSC), qui repose sur des composants issus de NetBeans, permet de se pencher d'avantage sur la couche web de l'application. Des modules permettent de créer une connexion à des *Session Beans* ou des services web, d'éditer une page HTML de façon graphique… Il est lui aussi totalement libre d'utilisation. On peut le télécharger à l'adresse : http://developers.sun.com/prodtech/javatools/jscreator/index.jsp.

Fonctionnalités de NetBeans 5.5

NetBeans propose un support intéressant de Java EE 5. On peut s'en rendre compte lors de la création d'un projet ; on se voit proposer un support des applications web, EJB, Web Services, SOA…

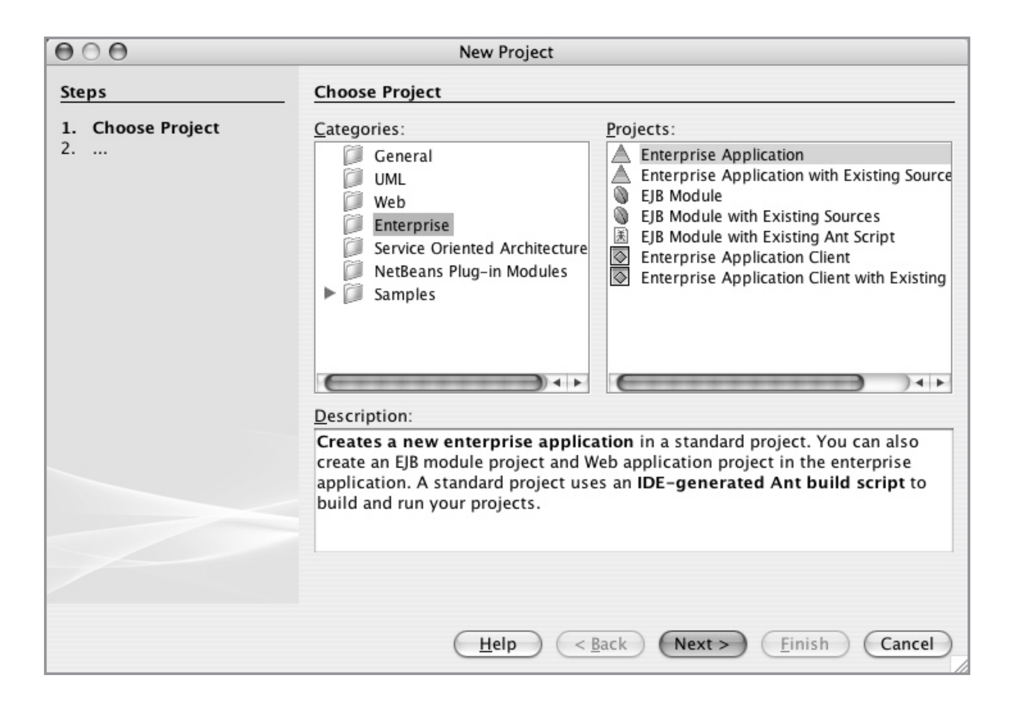


Figure 10.7 — File → Create New Project …

De plus, très peu de configuration est nécessaire car NetBeans automatise beaucoup d'éléments comme le déploiement, l'écriture des fichiers de configuration… Certes, cette automatisation poussée à l'extrême risque de ne pas convenir à tous les programmeurs, mais le gain de temps n'est pas négligeable lorsque l'on maîtrise cet éditeur.

Prenons l'exemple d'un projet où nous souhaitons programmer une application avec un module EJB 3, ainsi qu'une partie Web. La création d'un projet de type « Entreprise-Application » s'impose alors (fig. 10.7). Ce type de projet va regrouper différents sous-projets afin d'avoir une organisation claire (fig. 10.8). Ils seront ensuite déployés sous forme d'un fichier .ear (fig. 10.9).

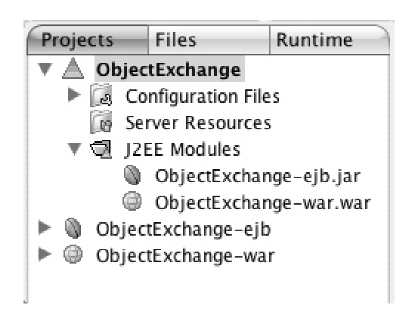


Figure 10.8 — Le projet Entreprise est automatiquement lié avec des modules EJB, Web…

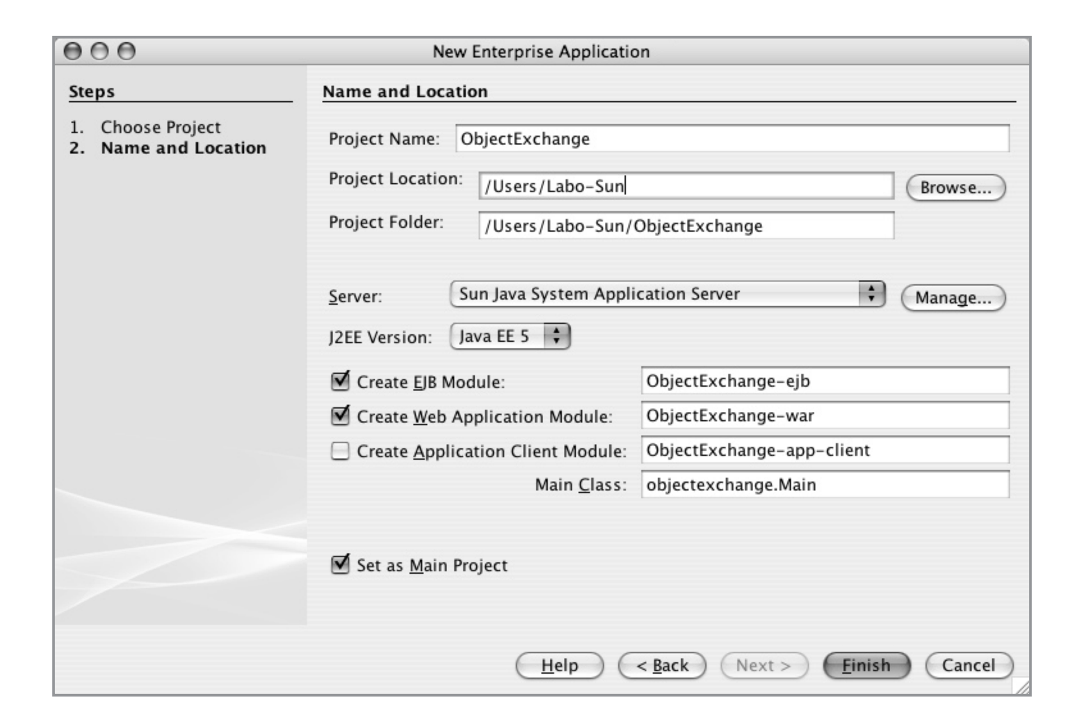


Figure 10.9 — Créations des modules d'une application d'entreprise

Support des applications externes

L'intégration d'éléments externes à l'IDE, comme le serveur d'applications ou la base de données, est un élément pratique non négligeable. L'onglet « Runtime » permettant d'accéder à ces outils externes, ainsi qu'à d'autres services (fig. 10.10).



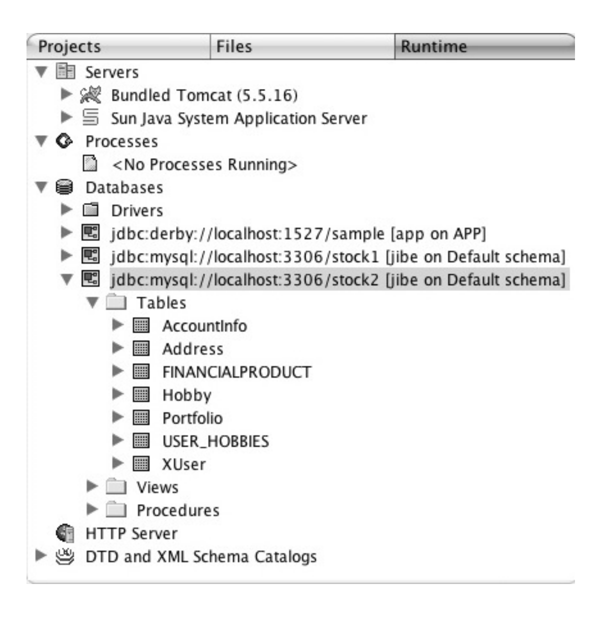


Figure 10.10 — Accès directs aux serveurs d'application et à la base de données à partir de NetBeans

Le serveur web Tomcat est inclus afin de pouvoir programmer et déployer rapidement des applications web simples (servlets, JSP et services web).

SunAppServer9 correspond à SJSAS, qui, quant à lui, prend en charge toutes les applications entreprises J2EE 1.4 et Java EE 5. Cette intégration au sein de NetBeans a aussi l'avantage de proposer toutes les fonctionnalités de son interface web d'administration.

On peut remarquer qu'un support de JBoss existe, pouvant être utilisé pour les applications J2EE 1.4. Celui-ci est cependant minimal, ne permettant que de démarrer, stopper, et effectuer du débogage.

Débogage d'un EJB

Comme dans la plupart des IDE, le débogueur permet d'exécuter pas à pas les applications. Cela fonctionne aussi bien sur un client riche qu'avec des EJB ou des applications web.

Au préalable, nous plaçons dans notre code des points d'arrêts (*breakpoints*), qui sont représentés par des petits carrés dans la barre affichant le numéro des lignes (fig. 10.11).

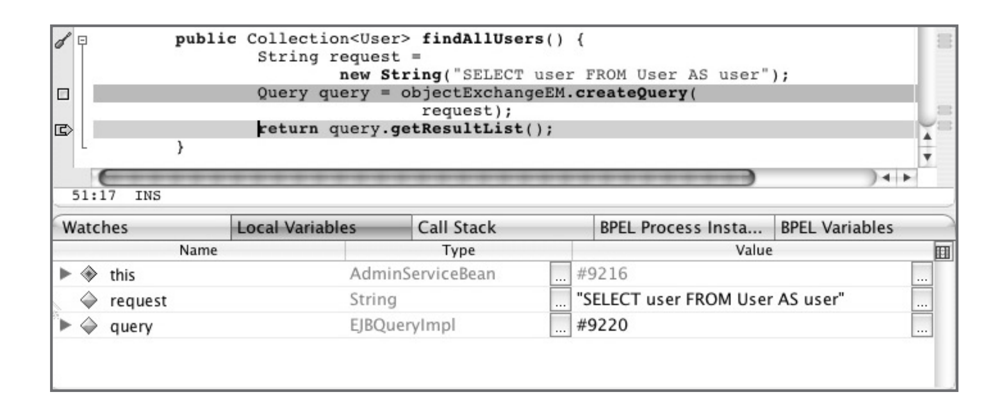


Figure 10.11 — Le placement de points d'arrêts permet de mettre en pause l'exécution du code

Pour utiliser le mode debug, effectuez un clic-droit sur le projet dans l'onglet « Projects », puis exécutez l'application en sélectionnant « debug project » (fig. 10.12).

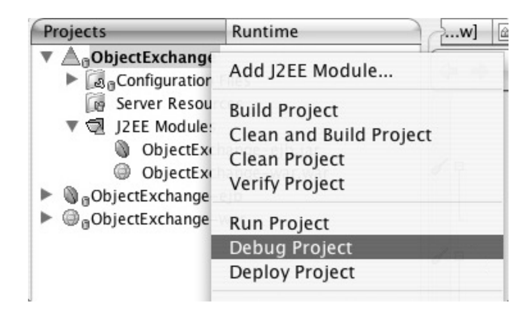


Figure 10.12 — Exécutions d'un projet en mode debug

Lorsque l'exécution de l'application arrivera sur les instructions marquées, l'application sera mise en pause, et le code Java correspondant s'affichera. La vue nommée « Local Variables » détaillera les informations contenues en mémoire. Les différents boutons de la barre d'outils (fig. 10.13) du débogueur pour continuer la lecture de l'application.

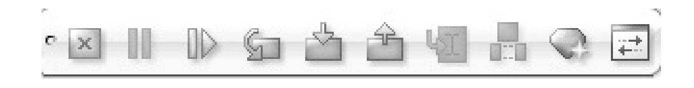


Figure 10.13 — La barre d'outils du débogueur

10.3 UNE SOLUTION ALTERNATIVE

10.3.1 JBoss

Historique



Figure 10.14 — Logo JBoss

JBoss est devenu l'un des serveurs d'applications les plus populaires. Cette plateforme sous licence GPL[1], est certifiée J2EE 1.4 mais propose tout de même une implémentation des spécifications des EJB 3. Celui-ci fournit, de plus, une large gamme de produits pour la gestion du *clustering*, de la mise en cache, ou encore de la persistance. Tout cela est gratuit et repose sur le travail de la communauté de développeurs.

La compagnie mettant à disposition ce serveur s'appelle « JBoss Inc » depuis avril 2006 racheté par Red Hat. Le siège social de la compagnie se trouve à Atlanta (États-Unis) où Marc Fleury, un français expatrié, conduit le projet.

Parmi les projets Open Source gravitant autour de JBoss, on compte Hibernate (un célèbre moteur de persistance, utilisé pour l'implémentation de la persistance des EJB 3), JBoss Seam, JBoss Transactions, JBoss Messaging… Tous ces produits sont fournis dans la suite JBoss Entreprise Middleware Suite (JEMS). Il a notamment été l'un des premiers à proposer une implémentation fonctionnelle d'EJB 3.

L'autre avantage de JBoss, qui a fait sa réputation, est également la qualité de sa documentation disponible à l'adresse suivante : http://www.jboss.com/docs/index

La version 3 de JBoss propose une implémentation de Java EE 1.4. Celle actuellement en développement à l'écriture de ce livre, la version 4, ajoute une implémentation de Java EE 5.

Installation

Le téléchargement se fait à partir du site JBoss (www.jboss.org), dans la rubrique « Download → JBoss Application Server ».

1. GPL : GNU Public Licence.

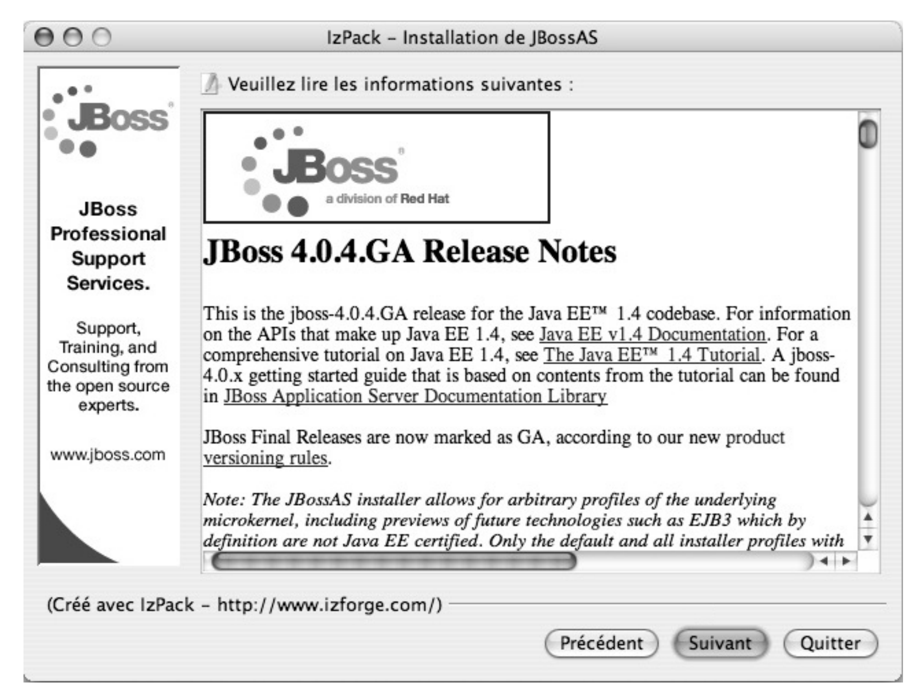


Figure 10.15 — Interface d'installation de JBoss

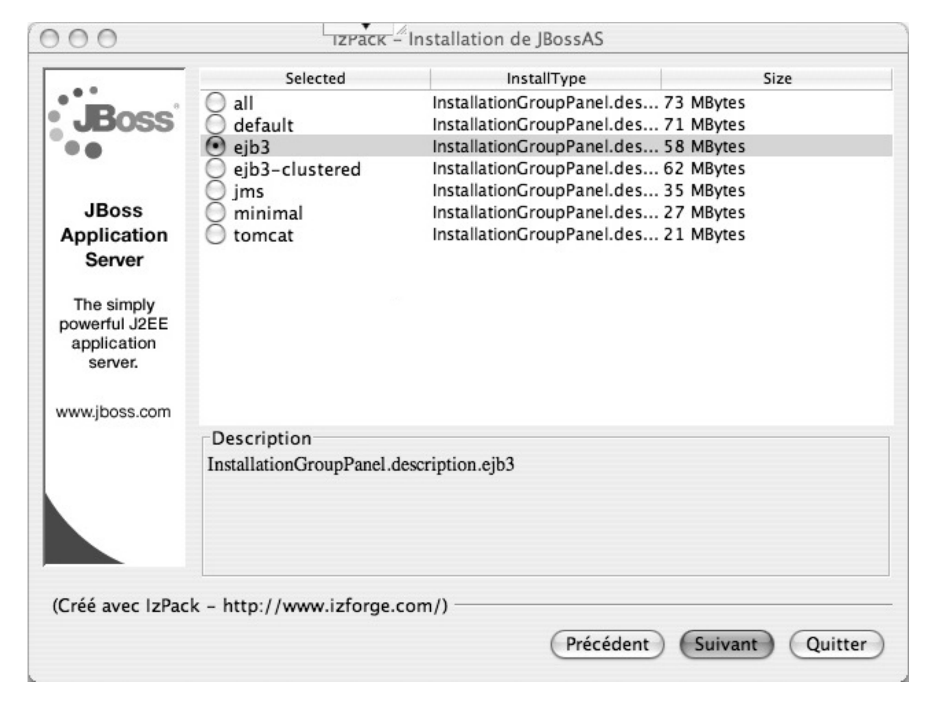


Figure 10.16 — Sélection de la configuration

Le mode d'installation sélectionné définira les libraires installées, et les modules qui seront chargés lors de l'exécution du serveur. L'installation de type « all » est la plus complète, cependant la mémoire utilisée et le temps de démarrage s'en ressentiront.

On peut regretter que son administration soit complexe. Celle-ci repose uniquement sur des fichiers XML et aucune interface d'administration conviviale n'a été proposée jusqu'à ce jour (fig. 10.17).

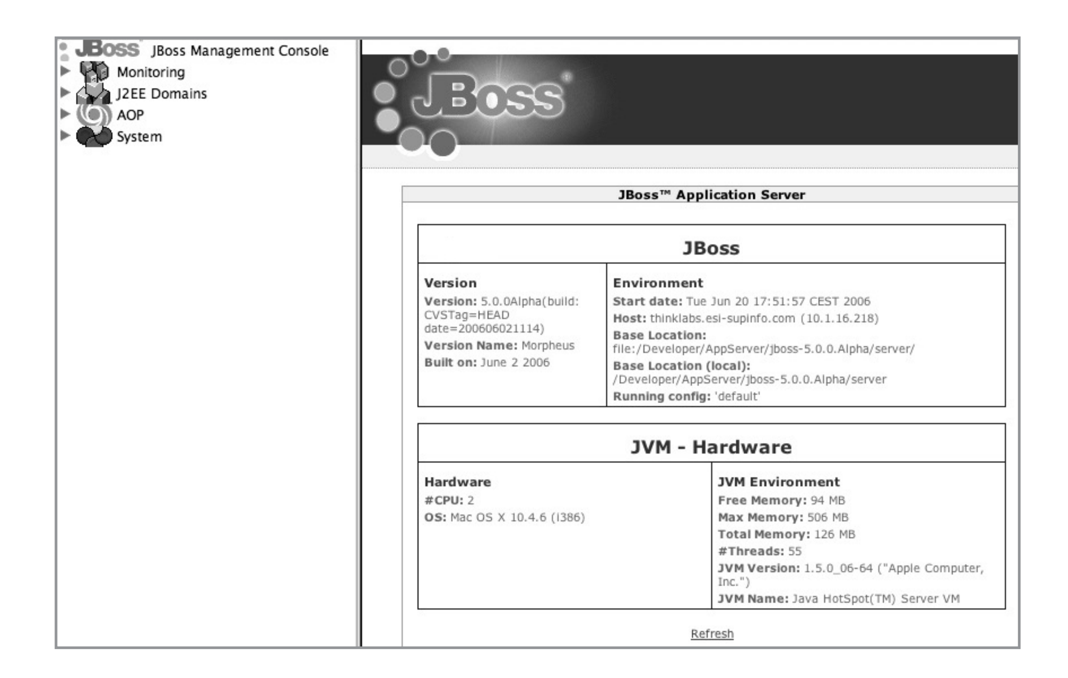


Figure 10.17 — Interface d'administration de JBoss

Malgré tout, JBoss propose depuis peu une suite nommée JBoss Operations Network qui offre, en plus du serveur d'applications, une interface graphique de configuration (http://www.jboss.org/services/jbossnetwork).

Configuration de JBoss dans le système

Afin d'assurer un bon fonctionnement de JBoss, il est conseillé de définir une variable d'environnement nommée « JBOSS_HOME » sur votre système.

- Sous Windows : « Démarrer → Panneau de configuration → Système → Avancé → Variables d'environnements », puis créez une nouvelle variable Système nommée JBOSS_HOME, pointant vers le répertoire d'installation de JBoss (fig. 10.18).

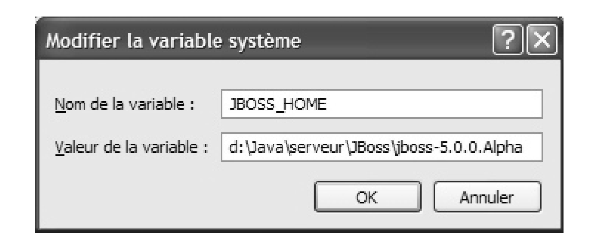


Figure 10.18 – Configuration de la variable d'environnement

- Sous Linux, Mac ou Solaris :

```
export JBOSS_HOME=/usr/local/jboss
```

Structures des dossiers de JBoss

L'organisation des répertoires de JBoss est un élément important à connaître afin de pouvoir le configurer, déployer une application, ou réaliser toute autre opération sur le serveur. En effet il n'existe pas d'environnement de développement, ou d'interface d'administration permettant de gérer tout cela automatiquement.

Lors de l'installation, les éléments créés sont des scripts de démarrage pour JBoss, des fichiers de configuration XML, ainsi que des librairies Java. Leur hiérarchie est présentée figure 10.19.

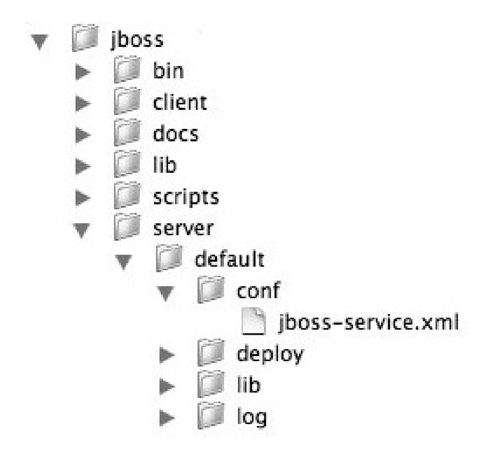


Figure 10.19 – Structure des répertoires de JBoss

Dans le répertoire *bin*, vous trouverez la liste de scripts nécessaires au démarrage et à la fermeture de JBoss, avec « run » et « shutdown », permettant respectivement de démarrer et d'arrêter le serveur. Les scripts pour environnement Unix, Mac et Solaris ont l'extension « .sh » tandis que les scripts pour Windows ont pour extension « .bat ».

L'appel du script « run » permet de démarrer différentes configurations du serveur avec le paramètre « -c <nom de la configuration> ». Ces configurations correspon-

dent aux sous-répertoires de « server ». On peut donc ajouter un répertoire « server/test », pour le démarrer avec « run –c test ». Ces configurations internes à JBoss permettent de disposer à la fois d'une configuration pour la mise en production, une autre pour les tests (avec une configuration avancée des logs, par exemple), une autre pour les applications web, une pour les EJB…

Chacune de ces configurations contient trois éléments indispensables :

- Le répertoire « deploy » qui contient tous les services, data sources et applications déployées.
- Le fichier de configuration « conf/jboss-service.xml » qui permet de paramétrer le chargement des services, des librairies, d'initialiser le gestionnaire de log et des autres fichiers de configuration.
- Le répertoire « lib » qui est nécessaire pour charger les librairies tierces, comme les pilotes d'accès à une base de données par exemple.

Serveur web

Dans la configuration minimale, JBoss contient un serveur web écoutant par défaut sur le port 8080. Ce dernier est basé sur TomCat, un autre serveur d'applications très répandu (permet d'utiliser seulement des servlets/JSP). Tous les fichiers de configuration de celui-ci se trouvent dans le répertoire « server/<configuration>/deploy/jbossweb-tomcat55.sar »

La configuration du conteneur web, comme le numéro du port d'écoute, se fait dans le fichier de configuration « server/<configuration>/deploy/jbossweb-tomcat55.sar/conf/web.xml ».

Pour plus d'informations sur son administration, la documentation de TomCat est disponible à l'adresse suivante :
http://tomcat.apache.org/tomcat-5.5-doc/index.html.

Conteneur d'EJB

Le conteneur d'EJB ne nécessite pas de configuration particulière. Le plus important est de spécifier les archives jar essentielles. Lors de vos développements, il est utile d'avoir les librairies suivantes dans le classpath du compilateur et du serveur d'applications :

- jboss-ejb3.jar, dans server/default/deploy/ejb3.deployer
- jboss-ejb3x.jar, dans server/default/deploy/ejb3.deployer
- ejb3-persistence.jar, dans server/default/lib
- jboss-j2ee.jar, dans server/default/lib

Pour le client, certaines librairies sont également essentielles afin d'accéder aux sessions beans :

- jboss-ejb3.jar, dans server/default/deploy/ejb3.deployer
- jboss-ejb3x.jar, dans server/default/deploy/ejb3.deployer

- jboss-j2ee.jar, dans server/default/lib
- jboss-serialization.jar, dans server/default/lib
- jbossall-client.jar, dans client
- jboss-aspect-jdk50-client.jar, dans client
- jboss-aop-jdk-client.jar, dans client
- hibernate3.jar, dans server/default/lib
- jboss-remoting.jar, dans server/default/lib
- commons-httpclient.jar, dans server/server/default/lib
- antlr, dans server/default/lib

Cette organisation des librairies au sein des répertoires de JBoss peut changer en fonction des versions, en intégrant, par exemple :

- des plugins et fonctionnalités avancées,
- un système *clustering* et *load-balancing*,
- un système de déploiement distribué (*farming*).

10.3.2 Eclipse

Historique

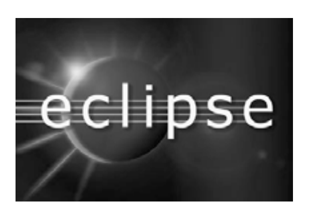


Figure 10.20 — Logo Eclipse

Eclipse IDE peut être comparé à NetBeans dans le sens où ils sont ouverts à la communauté des développeurs, entièrement écrits en Java, et fortement modulaires. Cependant, cet outil et la communauté gravitant autour, sont « sponsorisés » par des entreprises comme IBM, Borland, Oracle…

À l'origine, Eclipse correspond à la mise sous licence GPL de Visual Age For Java, un environnement de développement créé par IBM. En 2006, Eclipse est l'IDE du monde Java le plus utilisé, et il bénéficie de nombreux *plugins* supportant le C/C++, le PHP, l'accès aux bases de données…

La force d'Eclipse réside dans le fait que celui-ci est beaucoup plus qu'un simple IDE c'est une plate-forme complètement personnalisable et évolutive.

Ce succès réside dans la personnalisation d'Eclipse grâce à l'ajout de *plugins* permettant d'étendre les fonctionnalités de base.

Lorsque vous développez des applications qui utilisent les technologies J2EE et JBoss, vous pouvez utiliser le plugin WTP (*Web Tools Plateform*). Il existe aussi le *plugin* JBossIDE qui propose un support intéressant de JBoss. D'autres *plugins* comme Hibernate Tools 3.1 sont aussi intéressants et nous vous conseillons de les essayer. Cependant nous allons uniquement détailler WTP qui couvre l'ensemble de nos besoins.

Le plugin WTP

WTP (*Web Tools Platform*) est un *plugin* qui regroupe un ensemble d'outils destinés à développer des applications J2EE. Parmi les outils que vous serez le plus amené à utiliser, vous retrouverez des éditeurs HTML, JavaScript, CSS, JSP, XML, un explorateur de services web, un outil d'accès aux bases de données…

Pour ceux qui souhaitent utiliser ce *plugin*, vous pouvez le télécharger à l'URL suivante : http://www.eclipse.org/webtools/. Vous avez également la possibilité d'installer celui-ci par l'intermédiaire de *l'update manager* intégré à l'Eclipse *via* l'URL suivante : http://download.eclipse.org/webtools/updates/.

La version 1.5 de WTP se décompose en plusieurs sous-projets :

- *Web Standard Tools* (WST) : regroupe des outils d'harmonisation d'architecture, support des Web Services (Axis 2.0), support du framework Validation.
- *J2EE Standard Tools* (JST) : support de J2EE 1.5, EJB 3, JavaServer Faces, Web Services, Server Runtime.

Une fois ce plugin actif au sein d'Eclipse, vous avez la possibilité de créer de nouveaux types de projet. Lorsque vous développer des applications web vous allez généralement choisir un projet de type « Web/Dynamic Web Project » (fig. 10.21).

L'intérêt de WTP est qu'il génère toute la structure (le squelette) de votre projet. Celui-ci contient alors un répertoire source (src) et un répertoire web (WebContent).

WTP 1.5 intègre des outils très intéressants pour la création de vos Entity et Session Beans. Si nous reprenons notre Entity Bean `User`, par exemple, au lieu de créer une classe Java simple, nous allons utiliser WTP en choisissant le type « Entity » de la catégorie « Java Persistence » (fig. 10.22).



Figure 10.21 – Création d'un projet web avec WTP



Figure 10.22 – Création d'un entity avec WTP (étape 1)

L'assistant de création de classe s'affiche alors (fig. 10.23), permettant de paramétrer notre Entity Bean.

Figure 10.23 — Création d'un entity avec WTP (étape 2)

WTP intervient surtout au niveau de l'éditeur de code puisque de nouvelles fenêtres seront spécifiques à ce nouveau type « Entity ». Après validation de l'assistant, un écran de création de connexion (fig. 10.24) permet de lier le Bean à la base de données existante.

Les champs de l'assistant précédent sont vides lors de la première création de l'Entity, il faut donc définir la connexion *via* le lien « Add connections… ».

Add Java Persistence
JPA Project Content
Enter the data required to generate the JPA project content.
Database Settings
Connection:
Schema:
(Note: Must have active connection to select schema)
Add connections...
Reconnect...
Classpath Configuration (Optional)
Configure the project build path...
Packaging Settings
Create persistence.xml
Persistence version: 1.0
Persistence provider:
Persistence unit name:
Cancel Finish

Figure 10.24 – Configuration de la persistance (étape 1)

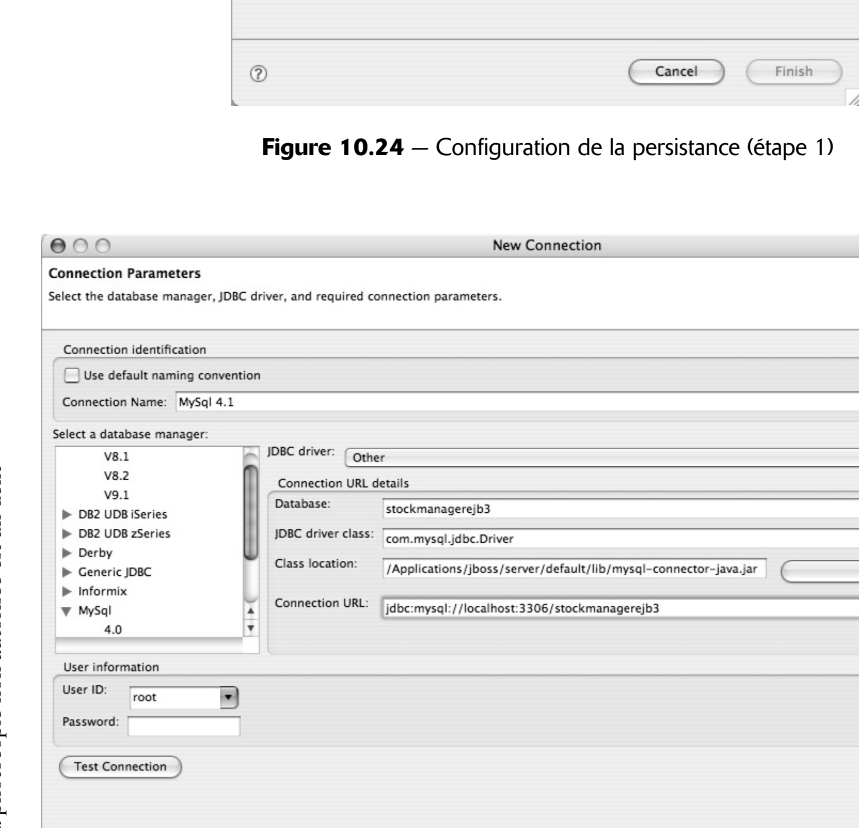


Figure 10.25 – Configuration de la persistance (étape 2)

Il faut, dans ce nouvel assistant (fig. 10.25) définir le type de base de données, le nom de la base de données, le pilote à utiliser ainsi que son emplacement. Lorsque l'on valide, cet assistant cela remplira les champs de l'assistant précédent.

Figure 10.26 — Configuration de la persistance (étape 3)

Après validation des informations, on accède à l'éditeur de code. Une nouvelle perspective est alors accessible « Windows > Show Perspective > Other … > Java Persistence ». Cette perspective se compose de trois vues supplémentaires : « Database Explorer », « Persistence Outline » et « Persistence properties ».

La vue « Database Explorer » (fig. 10.27) permet de naviguer dans votre base de données existante (celle-ci doit être démarrée) à partir des informations de connexions renseignées dans les assistants précédents.

Supposons que l'on reprenne les propriétés de la classe `User` et que nous souhaitions mettre à jour les getters/setters. Grâce à la vue « Persistence Properties », nous allons pouvoir recréer le *mapping* de manière très rapide.

Il faut tout d'abord définir à quelle table existante dans votre base de données votre classe devra être *mappée* (fig. 10.28).

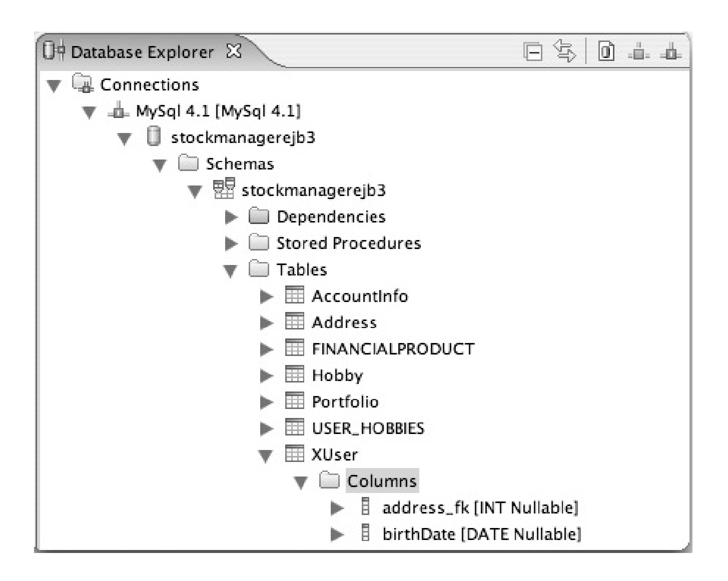


Figure 10.27 — Vue « Database Explorer »

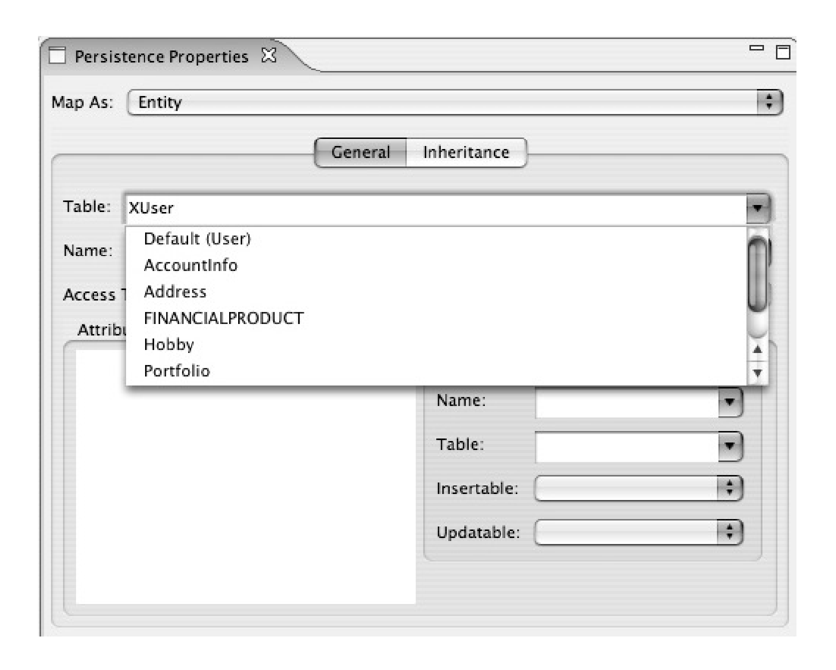


Figure 10.28 — Création du mapping (étape 1)

Il suffit, ensuite, de définir comment est *mappée* la relation dans la zone « Map As » puis le Bean auquel la relation fait référence (ce champ est rempli de manière automatique). La dernière chose que nous allons faire, pour relier notre Entity Bean `User` et `Portfolio`, consiste à définir le propriétaire de la relation en sélectionnant dans « Mapped By » (fig. 10.29) le champ « user » faisant référence à notre `User` :

Figure 10.29 — Création du mapping (étape 2)

Cette vue génère les annotations suivantes :

```
@Entity
@Table(name="xuser")
public class User {
  //...
  @OneToMany(mappedBy="user")
  private Collection<Portfolio> portfolios;
  //...
```

La vue « Persistence Outline » (fig. 10.30) est utilisée pour symboliser tous les champs persistants de l'entité. Cette vue est créée à partir des annotations présentes dans le code source des EJB.

Ces vues ne sont pas les seules nouveautés des Entity Beans version 3. Un nouveau menu apparaît désormais, lorsque l'on fait un clic droit sur le projet : le menu « Java Persistence ». À partir de ce menu on a deux options possibles : « Generate Entities … » et « Generate DDL … ». La première de ces options utilise le principe de la retro-conception (*reverse engeenering*), à partir d'une base de données. On peut ainsi recréer les Entity Beans très rapidement. La seconde option permet de générer l'ordre SQL de génération de base de données. En effet, nous ne sommes pas obligés d'utiliser une base de données existante pour se servir de ce *plugin*.



Figure 10.30 — Vue « Persistence Outline »

Dans Eclipse, le déploiement et le *packaging* peuvent soit se faire de manière intégrée au *plugin* WTP, soit grâce à l'utilisation de Ant qui permet d'automatiser ces tâches.

10.4 SEAM : UN FRAMEWORK D'AVENIR

Nous avons vu dans les sections précédentes que les outils (IDE principalement) étaient d'une grande aide pour les développeurs. Même s'ils aident à écrire le code de l'application plus facilement et plus rapidement, ils n'aident pas forcément à la connexion entre les composants et à l'architecture de l'application.

Lorsque le standard servlet/JSP a été lancé, Struts s'est imposé comme un des meilleurs *frameworks* pour la mise en place du modèle MVC. La nouvelle spécification Java EE 5 intègre deux grandes fonctionnalités JSF et EJB 3. Cependant, elle n'impose rien en matière de connexion entre ces composants (à part l'injection possible d'EJB dans un *Managed Bean* JSF). Seam est donc là pour remédier à ce manque !

10.4.1 Présentation

Seam est un *framework* Open Source développé par la société JBoss (récemment rachetée par Red Hat). Il permet d'unifier les modèles de composants JSF (*Managed Beans*) et les EJB 3 en éliminant tout code inutile nécessaire à la connexion entre JSF et EJB 3. Les développeurs peuvent alors se focaliser complètement sur le code métier.

Le grand changement vient du fait, qu'avec Seam, il n'existe plus de réelle différenciation entre les composants « vues » et les composants « modèle » du concept MVC. De plus, Seam est totalement orienté EJB. Il est même possible de créer une application où tous les composants sont des EJB ! C'est un retour complet sur cette technologie qui était souvent montrée du doigt à cause de sa lourdeur. Cependant, EJB 3 change la donne et le point de vue des développeurs sur cette technologie.

Seam offre, en plus des contextes fournis par la spécification (request, session, application), deux nouveaux contextes : conversation et processus métier. Ces contextes sont totalement orientés vers la logique métier de l'application. De plus, ils optimisent la gestion des états de l'application. En effet, pour de nombreux développeurs, la gestion des attributs dans la session était souvent délaissée, et cette approche engendrait très souvent de nombreux bogues ou problèmes de mémoire. Seam le gère maintenant pour vous…

Nous avons énormément parlé d'injection dans l'ensemble de cet ouvrage et JSF intègre la possibilité cette fonctionnalité. Toutefois, dans certaines applications, certains composants peuvent partager et modifier les mêmes données. Seam intègre pour cela le mécanisme de « bijection » : il permet de connecter des variables contextuelles à des attributs de composants.

De nombreuses autres fonctionnalités sont de plus disponibles comme l'intégration d'un modèle de processus métier (BPM, *Business Processing Management*), la configuration par annotation, les tests facilités (on ne travaille qu'avec des POJO)…

> **Remarque :** ces quelques pages ont pour but de vous présenter Seam et nous vous conseillons fortement d'aller voir et le tester à partir du site officiel : http://www.jboss.com/products/seam.

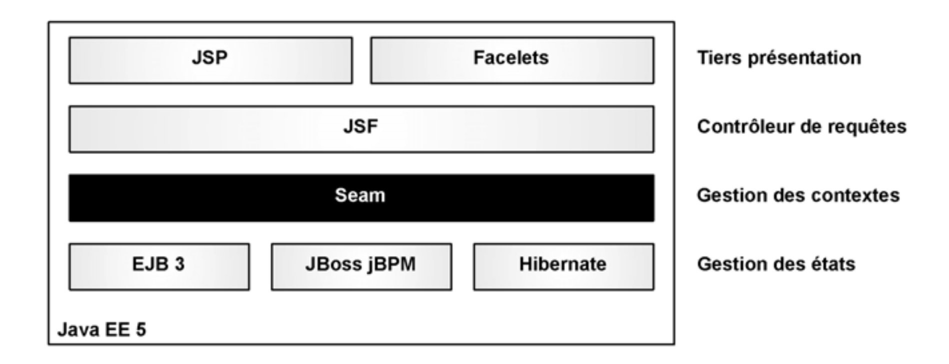


Figure 10.31 — Architecture d'une application Seam

Comme le montre la figure 10.31, Seam est le point central de toute application l'utilisant. Il connecte la vue et les modèles de façon simple et transparente.

10.4.2 Exemple

Nous allons présenter un exemple très simple de gestion de contacts. Seam étant un *framework* passionnant et très complet, nous ne pourrons l'expliquer en détail car un ouvrage complet pourrait lui être dédié.

Notre application permet d'ajouter, modifier, supprimer et lister des contacts. Pour simplifier son développement, nous avons créé une boucle entre l'étape d'ajout, la liste et la modification.

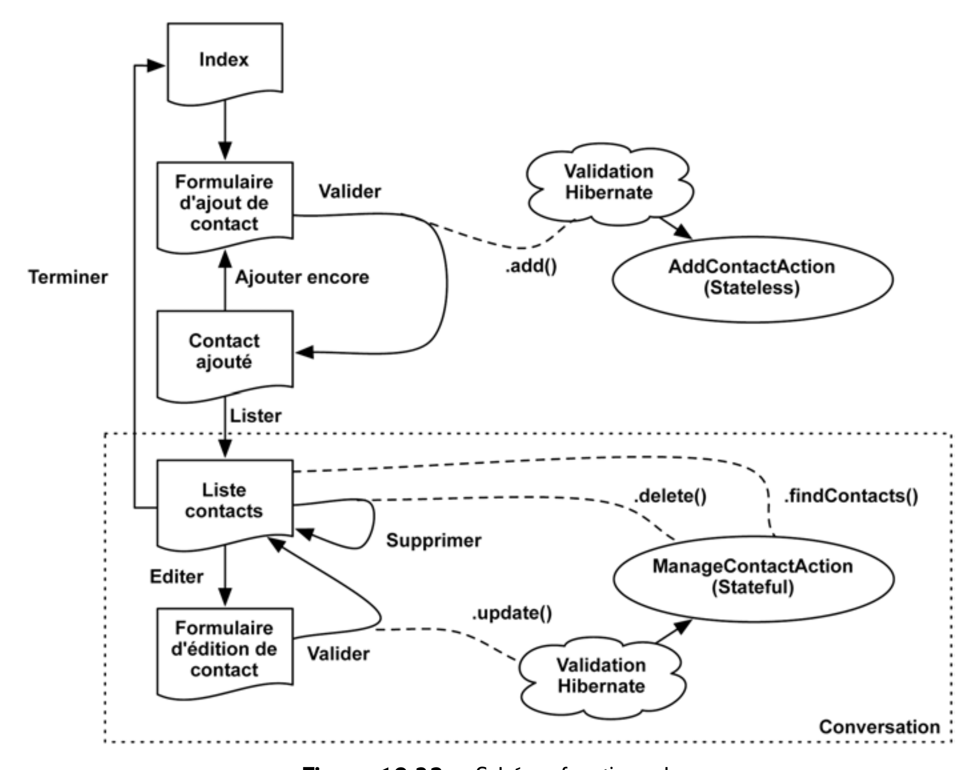


Figure 10.32 — Schéma fonctionnel

Le schéma de la figure 10.32 intègre la navigation entre les pages et les actions affectées liées à cette navigation. L'application utilise deux Session Beans :

- `AddContactAction` : gère l'ajout d'un contact.
- `ManageContactAction` : gère la liste, modification et suppression de contacts.

Voici la classe `AddContactAction` :

```
@Stateless
@Name("addContact")
public class AddContactAction implements AddContact {

  @In
```

```
  @Valid
  private Contact contact;

  @PersistenceContext(unitName = "contactUP")
  private EntityManager em;

  @Logger
  private Log log;

  @IfInvalid(outcome=Outcome.REDISPLAY)
  public String addContact() {
    List existing = em.createNamedQuery("findContactByIdentity")
        .setParameter("identityCardNumber",
            contact.getIdentityCardNumber()).getResultList();
    if (existing.size() == 0) {
      em.persist(contact);
      log.info("Contact #{user.firstName} #{user.lastName} "
        +"ajouté avec succès");
      return "/addedContact.jsp";
    } else {
      FacesMessages.instance().add(
          "Contact #{user.firstName} #{user.lastName} "
             +"existe déjà ! (même numéro d'identité)");
      return null;
    }
  }

  public String finish() {
    return "/listContact.jsp";
  }

  public String addAgain() {
    return "/addContact.jsp";
  }
}
```

Les annotations `@Name`, `@Logger`, `@IfInvalidn` et `@In` sont liées à Seam alors que `@Valid` est lié à Hibernate.

- L'annotation `@Name` permet de définir un nom unique pour le composant Seam. Il pourra alors être appelé *via* le nom donnée dans JSF (« #name.methode »).
- L'annotation `@In` indique que la propriété doit être injectée par Seam. La variable du contexte Seam est nommée « contact », correspondant au nom de la variable d'instance
- L'annotation `@Valid` indique que l'objet peut être validé récursivement par le validateur Hibernate.
- L'annotation `@IfInvalid` demande à Seam de lancer une validation des objets annotés avec `@Valid` avant d'appeler la méthode d'action. Si un problème survient, une redirection est faite automatiquement. Dans notre cas, la page du formulaire est réaffichée avec les messages d'erreurs liés à la validation.

Les méthodes `addContact()`, `finish()` et `addAgain()` retourne une chaîne de caractère. Celle-ci détermine la page qui doit être affichée après l'exécution de la méthode (et donc de l'action). Si la méthode retourne `null`, alors l'utilisateur reste sur la page courante.

La classe `Contact` représente un Entity Bean, toutefois nous avons dû ajouter certains éléments pour définir la validation d'une entité.

```
@Name("contact")
@Scope(SESSION)
@Table(name = "contacts")
@NamedQueries({
  @NamedQuery(name="findContactByIdentity", query="SELECT identityCardNumber "
    +"FROM Contact WHERE identityCardNumber=:identityCardNumber"),
  @NamedQuery(name="findAllContact", query="SELECT c FROM Contact c")}
)
public class Contact implements Serializable {

  private static final long serialVersionUID = 515766198330453074L;

  private int identityCardNumber;

  private String firstName;

  private String lastName;

  private Address address;

  private String email;

  private String phone;

  public Contact() {
    address = new Address();
  }

  @Id
  @NotNull
  public int getIdentityCardNumber() {
    return identityCardNumber;
  }

  public void setIdentityCardNumber(int identityCardNumber) {
    this.identityCardNumber = identityCardNumber;
  }

  @NotNull
  @Length(min = 2, max = 20)
  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String name) {
    this.firstName = name;
  }
```

```
  @NotNull
  @Length(min = 2, max = 20)
  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  @NotNull
  @Embedded
  public Address getAddress() {
    return address;
  }

  public void setAddress(Address address) {
    this.address = address;
  }

  @NotNull
  @Email
  public String getEmail() {
    return email;
  }

  public void setEmail(String email) {
    this.email = email;
  }

  public String getPhone() {
    return phone;
  }

  public void setPhone(String phone) {
    this.phone = phone;
  }

  @Override
  public String toString() {
    return "Contact(" + firstName + " - " + lastName + ")";
  }
}
```

- Les annotations `@NotNull`, `@Length` et `@Email` sont spécifiques à Hibernate et définissent les contraintes liées à la validation de l'entité. Vous pouvez alors remarquer qu'il est vraiment très simple de définir une validation de type Email sur une propriété.
- L'annotation `@Scope` définit l'endroit où est enregistrée l'instance crée par Seam. Dans l'exemple précédent, elle est enregistrée dans la session.

Voici maintenant le code de la page JSP permettant d'intégrer le visuel de l'application.

```
...
<body>
  <f:view>
   <h:form>
       <h2>Ajouter un contact</h2>
     <table border="0">
       <s:validateAll>
         <tr>
           <td>Numéro d'identité (carte identité) :</td>
           <td>
             <h:inputText
             value="#{contact.identityCardNumber}" required="true"/>
           </td>
         </tr>
         <tr>
           <td>Prénom :</td>
           <td>
             <h:inputText value="#{contact.firstName}" required="true"/>
           </td>
         </tr>
         <tr>
           <td>Nom : </td>
           <td><h:inputText value="#{contact.lastName}" required="true"/></td>
         </tr>
         <tr>
           <td>Email : </td>
           <td><h:inputText value="#{contact.email}" required="true"/></td>
         </tr>
         <tr>
           <td>Ville : </td>
           <td>
             <h:inputText value="#{contact.address.city}" required="true"/>
           </td>
         </tr>
         <tr>
           <td>Numéro / Rue : </td>
           <td>
             <h:inputText value="#{contact.address.number}" required="true"/>
              
             <h:inputText
             value="#{contact.address.street}" required="true"/></td>
         </tr>
       </s:validateAll>
     </table>
     <h:messages/>
     <h:commandButton type="submit" value="Ajouter"
       action="#{addContact.addContact}"/>
   </h:form>
  </f:view>
 </body>
</html>
...
```

Dans cet exemple, la « TagLib » validateAll est lié à Seam. Elle permet de définir la portée de la validation et offre d'autres fonctionnalités. Cette page utilise les composants Seam addContact et contact déclarés auparavant.

10.4.3 Conclusion

La *framework* Seam possède de nombreux avantages et nous en avons présenté à peine un dixième ! Il existe entre autres un système pour l'internationalisation, une gestion automatique de l'enchaînement des pages (*pageflow*), une intégration des processus métier et appel de méthodes à distance (*remoting*) *via* AJAX (*Asynchronous Javascript and XML*).

C'est un outil formidable qui simplifie la connexion entre le modèle (EJB) et la présentation (JSF) de l'application. Sa compatibilité avec WEB 2.0 *via* JBoss Remoting et son système de *workspace* (permettant de développer des applications web multifenêtrées) lui ouvre un très grand avenir.

La version 1.0.1 a été annoncée en juin dernier (2006) et les premiers retours sont très positifs (même depuis les premières versions bêta) : la technologie semble donc stable et très robuste…

Va-t-on aboutir à un nouveau standard ? Cela est fort probable car une idée de JSR commence à sortir des cartons…

En résumé

La communauté Java étant tellement vaste, il existe, aujourd'hui, de plus en plus d'outils, d'IDE ou de *frameworks* parallèles ou complémentaires à Java EE. Le choix entre toutes ces possibilités n'en est que plus difficile, tous rivalisant et mettant en avant leurs avantages.

Dans tous les cas, ce n'est qu'une question de goût. Le couple JBoss/Eclipse n'est pas moins performant que le couple SJSAS/NetBeans recommandé par Sun, ou bien d'autre.

À côté de cette « guerre » des IDE, de nouveaux *framework* prometteurs se développent permettant ainsi de simplifier davantage la connexion entre les composants.

11

Cas pratique

Objectif

Il est toujours plus facile de s'appuyer sur un exemple complet lorsque l'on souhaite apprendre une nouvelle technologie. Nous avons, pour cela, essayé d'utiliser au maximum une continuité pour les différents exemples de l'ouvrage. Toutefois, il est difficile d'étudier des morceaux de code placés à différents endroits.

Nous commencerons donc par une étude complète du fonctionnel de l'application, puis par la modélisation et l'étude technique de celle-ci. Pour finir, nous présenterons une partie du code source de l'application. Le code source de l'application complète, sera, quant à lui, disponible en téléchargement à l'adresse suivante :
http://www.labo-sun.com/resource-fr-articles-1176-0-ejb3-applications-exemples.htm

11.1 L'APPLICATION « OBJECT EXCHANGE »

11.1.1 Présentation générale

Notre application « Object Exchange » doit permettre de gérer des locations d'objets entre utilisateurs. Bien que simple, à première vue, cette application nécessite une étude préalable complète. Comme dans tout projet professionnel de développement, une étude des besoins et des fonctionnalités est importante pour la réussite de celui-ci.

L'application doit regrouper, dans une base de données, les objets « louables ». Chaque utilisateur inscrit gère sa propre liste d'objets qu'il propose et peut louer des objets aux autres utilisateurs.

Ces objets peuvent être de n'importe quel type (livres, DVD, CD, magasines...). L'application doit permettre d'initialiser la location et de suivre celle-ci. Pour cela nous utiliserons un système d'état : attente de validation, annulée, prêté, attente de retour, retourné. Le changement d'état est réalisé par l'intervention des utilisateurs (fig. 11.1) :

- *Attente de validation* : c'est le premier état de la location (lorsqu'un utilisateur valide une demande de location).
- *Annulée* : lorsque le propriétaire n'accepte pas la location de son objet.
- *Prêté* : lorsque le propriétaire a prêté l'objet demandé, celui-ci change l'état de la location.
- *Attente de retour* : lorsque la date de rendu est dépassée, une alerte automatique s'affiche dans le tableau récapitulatif des locations.
- *Retourné* : une fois l'objet rendu, le propriétaire termine la location qui passe en état retourné.

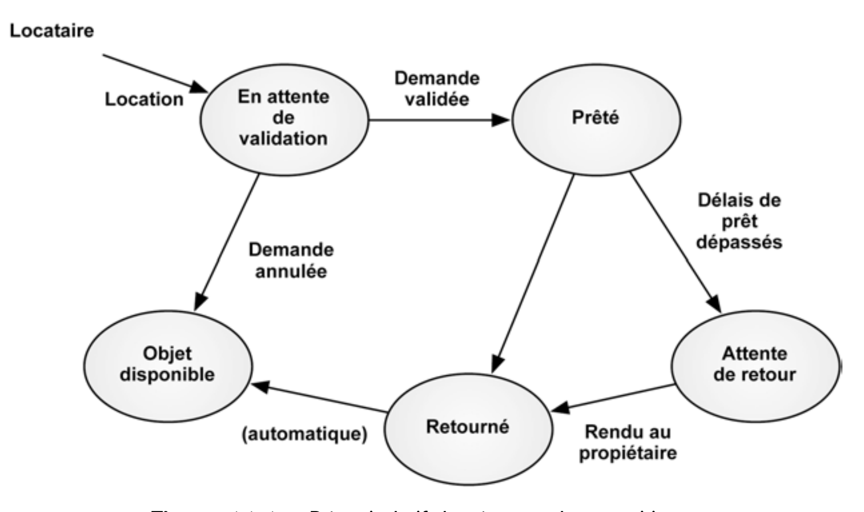


Figure 11.1 — Récapitulatif des états et des transitions

L'application doit permettre aux visiteurs (utilisateurs non inscrits) d'effectuer des recherches et de visualiser les objets disponibles en location. Ils devront cependant s'inscrire pour effectuer toute location.

L'inscription doit demander les informations de l'utilisateur (informations administratives) afin que son compte puisse être validé par un administrateur. Une fois qu'un visiteur est inscrit et validé par un administrateur, il peut s'authentifier et accéder à un ensemble plus important de fonctionnalités.

Un utilisateur authentifié peut ajouter, modifier, supprimer ou activer/désactiver des objets de sa liste au sein de l'interface d'administration de son compte. Il peut également modifier les informations le concernant (son profil). De la même façon qu'un visiteur, il peut consulter et rechercher des objets l'intéressant, et placer ceux qu'ils souhaitent emprunter dans un « panier virtuel ». Une fois ses choix faits, il peut valider son panier. Cette validation provoque un envoi de demande de prêt, par mail, au propriétaire de chaque objet demandé. Chaque location est enregistrée avec l'état « attente de validation ». Une fois l'objet passé en main propre, du propriétaire vers le locataire, celui-là a la charge de changer l'état de la location qui devient « En prêt ». Une fois la location retournée, le propriétaire doit terminer la location, permettant de modifier son état en : « Retourné ». L'objet est alors de nouveau disponible. Si le locataire n'a pas rendu l'objet avant la date limite de rendu, l'état de la location devient alors « Attente de retour ».

L'utilisateur authentifié peut également voir l'historique de ses locations et de ses prêts. Il peut donc à tout moment savoir si un de ses objets est loué et quel est son locataire. De la même façon, il peut vérifier qu'il n'est pas en retard sur le retour d'un objet qu'il a loué.

L'administration de l'application est réalisée par des utilisateurs spéciaux : les « super-utilisateurs ». Ceux-ci ont accès à une interface type d'administration où ils peuvent superviser l'ensemble des données de l'application : types et les catégories d'objets disponibles, les utilisateurs (activer/désactiver), les objets (activer/désactiver. Ceux à caractère violent, pornographique… sont interdits).

L'application doit également offrir un accès à d'autres applications clientes (de type C#, C++, Java…). Cet accès « externe » doit permettre de récupérer les informations relatives au compte d'un utilisateur sans passer par un navigateur web. Cela permet aux utilisateurs d'avoir un aperçu rapide sur l'état de leurs locations et prêts.

Nous venons de faire un tour général sur les fonctionnalités devant être intégrées dans notre application. Partant de cela, nous devons définir les différents cas d'utilisation de l'application. Nous exposerons ensuite nos choix techniques et, finalement, l'architecture de l'application.

11.1.2 Cas d'utilisation

Une fois les besoins établis, il faut ensuite étudier les technologies à utiliser. Il n'est pas nécessaire de définir précisément tous les cas d'utilisation, mais il faut se concentrer sur ceux qui peuvent poser problèmes. Ce sont ceux que nous détaillerons dans ce chapitre.

Page d'accueil

La page d'accueil est affichée lorsqu'un visiteur arrive à la racine de l'application. Elle présente l'application et liste les derniers objets ajoutés. C'est le point de départ de l'application. Les règles de navigations à partir de cette page sont multiples :

- Se connecter avec un compte existant *via* le formulaire d'authentification.

- Créer un nouveau compte à partir du lien : « Se créer un compte ».
- Rechercher un objet à partir du champ de recherche.

Créer un compte

La page de création de compte affiche un formulaire permettant à l'utilisateur de rentrer les informations relatives à son compte (nom, prénom, adresse, email). Une fois le compte ajouté à la base de données, un email est envoyé indiquant le mot de passe auto-généré pour que l'utilisateur puisse s'authentifier.

Les erreurs pouvant survenir lors de la création du compte sont les suivantes :

- Un ou plusieurs champs du formulaire ne sont pas valides. Dans ce cas, l'utilisateur est redirigé vers la page de formulaire afin de régler ces problèmes.
- Une erreur survient lors de l'ajout de l'utilisateur en base de données. L'utilisateur est alors redirigé par une page d'erreur générique détaillant l'erreur système et l'invitant à réessayer ultérieurement.

Authentification

Un utilisateur inscrit utilise son email et son mot de passe (envoyé par mail) pour s'authentifier sur le système. Pour cela il remplit les champs du formulaire de connexion.

Les erreurs pouvant survenir lors de l'authentification sont les suivantes :

- Les champs ont été mal remplis. L'utilisateur est alors redirigé vers la page d'authentification (page d'accueil) avec un message lui indiquant que les champs sont obligatoires.
- L'email et le mot de passe ne correspondent pas aux informations de la base de données. L'utilisateur est redirigé vers la page d'accueil avec un message lui indiquant que l'email et le mot de passe ne correspondent pas.

Recherche d'objets

N'importe quel utilisateur peut faire une recherche d'objet à partir de l'interface publique de l'application. Pour cela, l'utilisateur entre un critère de recherche par rapport au titre de l'objet et son type puis clique sur un bouton pour lancer la recherche. Les résultats de cette recherche sont affichés dans un tableau récapitulatif trié par type, puis par nom.

Seuls les utilisateurs connectés peuvent ajouter les objets dans leur panier afin de pouvoir les louer.

L'erreur pouvant survenir lors de la recherche est la suivante : aucun résultat n'est trouvé. Dans ce cas, l'application affiche un message indiquant qu'aucun résultat n'a été trouvé.

Demande de location

La demande de location est sans doute le cas d'utilisation le plus complexe de cette application. L'utilisateur voulant louer un objet doit d'abord l'ajouter à son panier virtuel. Pour cela, il peut effectuer une recherche. Dans le tableau des résultats, la dernière colonne contient un lien permettant d'ajouter l'objet au panier si l'utilisateur est inscrit et authentifié.

Le panier ne peut pas contenir deux fois le même objet, si l'utilisateur tente d'ajouter un objet déjà présent, la requête est annulée et l'utilisateur en est averti par un message d'information.

Une fois l'ensemble des objets à louer choisis, l'utilisateur peut valider son panier en cliquant sur le bouton « Louer les objets » qui se trouve sur la page récapitulative du panier. Cette action provoque l'envoi de mail à chacun des propriétaires des objets loués afin qu'ils acceptent ou refuse la location.

Les erreurs pouvant survenir lors de validation sont les suivantes :

- Un objet choisi a été loué entre-temps et n'est donc plus disponible. Dans ce cas, les objets non disponibles sont annulés et « grisés » dans le panier. L'utilisateur doit de nouveau valider les objets restants sans se préoccuper des objets non disponibles.
- Une erreur est survenue lors de la création des locations. L'utilisateur est redirigé vers une page d'erreur générique détaillant le problème technique et l'invitant à réessayer ultérieurement.

Ajouter un objet à sa liste

Un utilisateur authentifié a la possibilité d'ajouter des objets pouvant être loués à sa liste. Pour cela, il doit cliquer sur le lien : « Ajouter un objet » apparaissant dans l'interface membre. Un formulaire d'ajout d'objet apparaît regroupant l'ensemble des informations d'un objet : titre, type, détails, auteur, image de présentation, durée d'un prêt.

Les erreurs pouvant survenir lors de validation sont les suivantes :

- Un ou plusieurs champs du formulaire ne sont pas valides. Dans ce cas, l'utilisateur est redirigé vers la page de formulaire afin de régler ces problèmes.
- Une erreur survient lors de l'ajout de l'objet en base de données. L'utilisateur est alors redirigé par une page d'erreur générique détaillant l'erreur système et l'invitant à réessayer ultérieurement.

L'objet est automatiquement validé. Cependant un email est envoyé aux administrateurs qui doivent vérifier la qualité de l'objet et l'invalider en cas de nécessité.

Valider une location

Lorsqu'un objet est demandé pour location, le propriétaire reçoit une notification par email et doit valider la location. Pour cela, il s'authentifie et clique sur le lien de

l'interface membre : « Locations en attente de validation ». Il lui suffit de cliquer sur le bouton « Valider » pour accepter la location ou le bouton « Refuser » dans le cas contraire.

L'erreur pouvant survenir lors de validation est la suivante : une erreur survient lors de la modification de l'état de la location. L'utilisateur est alors redirigé par une page d'erreur générique détaillant l'erreur système.

De nombreux autres cas d'utilisation peuvent être détaillés ici. Cependant il n'est pas nécessaire de les décrire, ceux-ci ne posant pas de problème particulier. Nous récapitulons les différentes opérations affectées aux différents rôles dans la partie suivante.

11.1.3 Récapitulatif des rôles et opérations associées

À partir des données et des analyses faites précédemment, nous pouvons établir des diagrammes d'étude de cas. Les différents acteurs à prendre en compte sont les suivants : visiteur, membre, administrateur.

Visiteur

Les opérations accessibles à un visiteur sont les suivantes (fig. 11.2) : consulter la liste des documents, rechercher un document, créer un compte et s'authentifier.

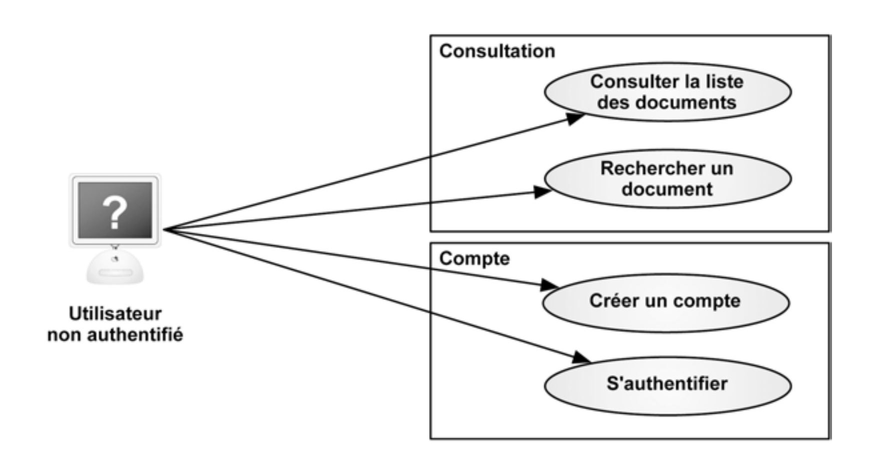


Figure 11.2 — Rôle visiteur

Membre

Il s'agit d'un visiteur authentifié sur le système. Voici les différentes opérations accessibles (fig. 11.3) : consulter les listes de documents, rechercher un document, gérer son panier virtuel (ajouter des objets, en supprimer, vider son panier...) et faire une demande de location (à partir du contenu du panier virtuel).

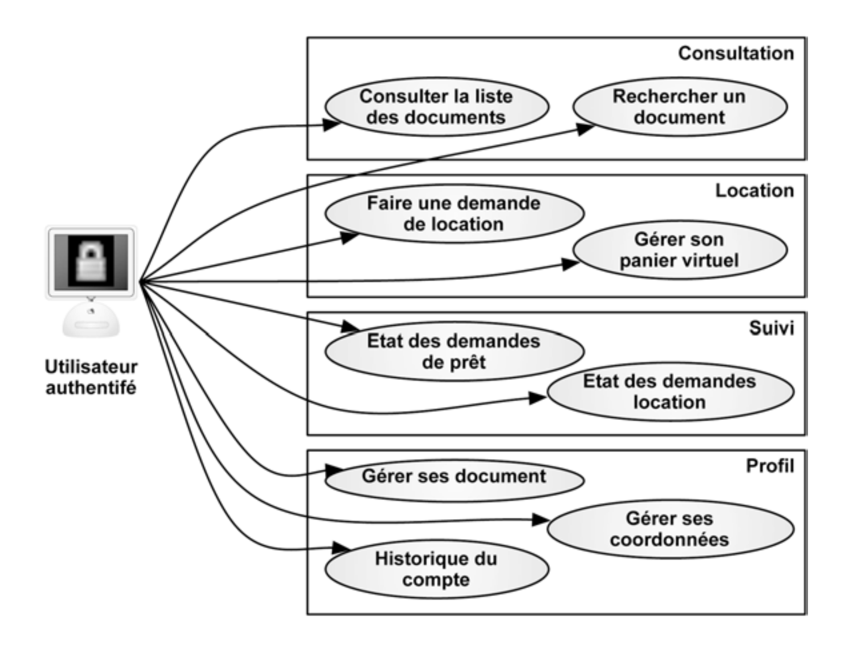


Figure 11.3 — Rôle membre

Il peut également visualiser l'état des objets prêtés et loués : suivre l'état d'une demande des prêts et suivre l'état d'une demande des locations.

Mais aussi gérer son profil : maintenir la liste des documents possédés, consulter l'historique du compte et spécifier ses coordonnées.

Administrateur

Il s'agit d'un compte particulier. En plus d'être un membre, il peut (fig. 11.4) : modifier tous les documents, désactiver les documents inappropriés, gérer des paramètres de l'application (type de documents…) et gérer les utilisateurs

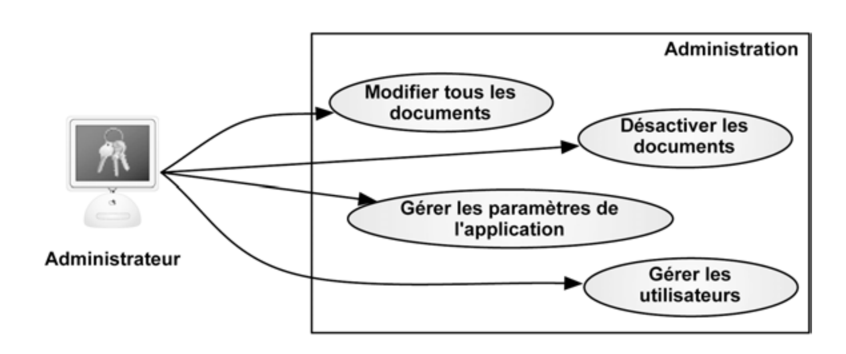


Figure 11.4 — Rôle administrateur

11.2 MODÉLISATION DE L'APPLICATION

11.2.1 Choix techniques

Avant même de réfléchir au code de l'application, il est essentiel d'étudier les différents environnements d'exécution disponibles. Dans une application d'entreprise, cet environnement est généralement composé de serveur(s) d'applications et de serveur(s) de bases de données.

Même si ces programmes sont souvent assimilés à des outils externes au système développé, il est important de les prendre en considération. En effet, ils ont des conséquences directes sur l'exécution de celui-ci.

Dans le cas du serveur d'applications, il est intéressant de connaître les possibilités de celui-ci. Est-il conforme aux spécifications utilisées ? Peut-il être *clustérisé* afin de faciliter la montée en charge dans votre application ?...

Nous avons choisi d'utiliser le serveur d'applications « Glassfish » (présenté au paragraphe 10.2.1) pour l'exécution de notre application, principalement pour son implémentation complète de la spécification Java EE 5. Cela permettra aussi de vous présenter plus concrètement celui-ci, au travers de cet exemple. Il est à notez que notre application s'exécutera également sur le serveur d'applications JBoss que nous avons utilisé tout au long de cet ouvrage.

De la même façon, le choix de la base de données est un critère très important. Elle centralise les données et gère les transactions. Les performances de celle-ci jouent considérablement sur l'application. Si la base de données est encombrée, l'application répondra très lentement aux requêtes des clients. Même si de nombreux paramètres existent au sein des SGBD pour les optimiser (mode transactionnel, cache, temps d'exécution de requête...), il est important d'étudier ceux-ci pour en sélectionner un qui correspond à vos besoins.

Notre application étant assez simple, nous avons choisi d'utiliser MySQL 5.0. Toutefois, celle-ci pourrait être remplacée par PostgresSQL ou Oracle XE sans problème.

11.2.2 Architecture physique

Une fois les choix techniques posés, il est important de réaliser la modélisation de l'application « en bloc ». Cette modélisation est le résultat d'une réflexion sur les différentes possibilités de l'application en termes de connectivité (client/serveur). Cette étude permet d'étudier les composants dans leur ensemble et non indépendamment.

Tout d'abord, les acteurs principaux de l'application sont :

- Les visiteurs peuvent voir la liste d'objets proposés.
- Les inscrits proposent des objets et demandent des locations. Ils sont avertis lorsqu'une date de retour de location approche ou est expirée.

- Les administrateurs gèrent les informations utilisables par l'application (catégories, types…).
- D'autres applications extérieures souhaitant récupérer des informations en lecture seule pour offrir un nouveau service à leur client.

Nous pouvons préciser dès à présent que l'application est accessible à partir de trois types d'applications :

- Une interface web (HTML) pour la majorité de l'application.
- Une application « client riche » SWING permettant d'avertir les clients concernant les alertes du système.
- Des applications externes qui se connectent *via* un web service fourni par le système (permettant par exemple de récupérer les derniers objets ajoutés).

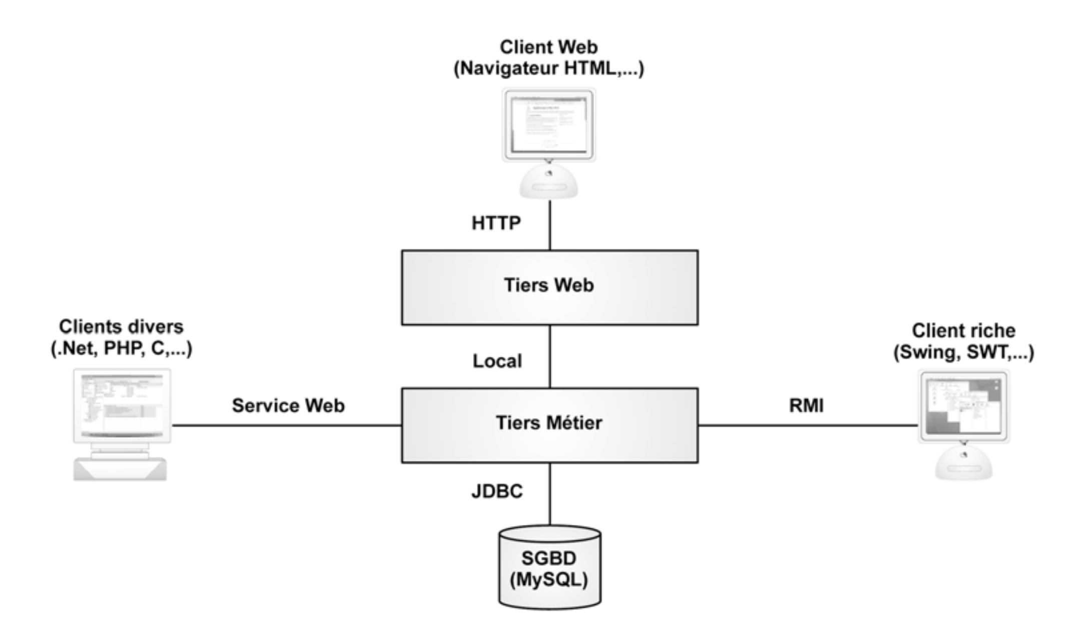


Figure 11.5 — Architecture physique

Le schéma de la figure11.5 présente l'architecture physique prévue pour le cas pratique. Nous retrouvons une architecture en couches séparées. La couche métier propose deux types d'accès :

- *Business to Consumer* (B2C) – C'est le système en lui-même avec l'interface web majoritairement et l'application Swing d'alerte.
- Business to Business (B2B) – Il est représenté par la possibilité aux applications externes de se connecter au système à partir du service web mis à disposition.

11.2.3 Analyse logicielle

Entités métiers

Pour répondre aux besoins fonctionnels énoncés précédemment, il faut maintenant définir les objets métiers de l'application. Nous allons donc définir des entités métiers permettant de rendre persistantes nos différentes données.

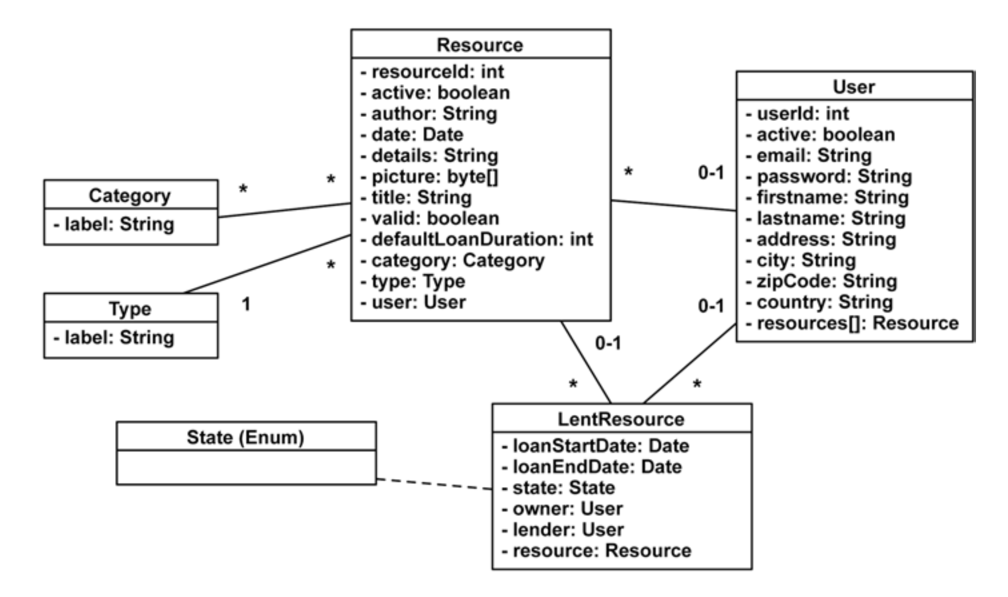


Figure 11.6 — Diagramme des entités métiers

Voici une description de chaque entité métier du diagramme de la figure 11.6 :

- `Type` représente les types de ressources. Les types sont : Livre, CD, DVD… Il permet de regrouper des ressources suivant un critère (le type).
- `Category` représente un critère permettant de définir le genre de la ressource. Ce concept est bien souvent utilisé dans les moteurs de recherche afin de limiter le nombre de résultat. On peut retrouver les catégories « Développement Java », « Administration Linux », « Educatif », « Entreprise »…
- `User` regroupe les informations relatives à un utilisateur du système. Comme dans de nombreux systèmes, on retrouve : le nom, le prénom, l'email, l'adresse (adresse, code postale, ville, pays). Il possède, de plus, une liste de ressources. Dans notre cas, cet objet métier est également utilisé pour gérer l'authentification des utilisateurs. À ce titre nous aurons donc besoin d'identifier l'utilisateur grâce à son email et un mot de passe. Un utilisateur pourra également être désactivé (banni) par un administrateur de la solution.
- `Resource` est l'entité centrale de l'application. Une ressource est composée d'un titre, d'un auteur, d'une date d'édition, de détails, du temps de prêt, d'une image (jaquette de l'album, page couverture du livre…). Une ressource pourra

également avoir les états : « valide » et « active ». Par défaut toutes les ressources seront valides, cependant l'administrateur se réserve un droit de regard sur les objets mis à disposition. Ce dernier pourra donc l'invalider s'il la juge inadéquate. L'utilisateur peut également, s'il le souhaite, activer ou désactiver ses ressources. Une ressource est liée à un `Type`, peut être répertoriée dans plusieurs `Category` et appartient à un seul utilisateur.

- `LentResource` définit un prêt. Cet objet est lié à une ressource, à l'utilisateur qui la loue et à l'utilisateur qui la prête. Il détient également les propriétés suivantes : date de début, date de fin et l'état du prêt (attente de validation, en prêt, attente de retour, retourné). Cet objet permet également de suivre les différents prêts qu'un utilisateur à fait mais également l'historique des prêts effectués sur une ressource.

Utiliser les Entity Beans

Nous avons vu précédemment les entités métiers de notre application. L'étape suivante est la définition du type de ces composants. Le type Entity Bean est le plus simple à identifier. Un Entity Bean représente une entité dans l'application (un compte bancaire, une personne, un objet ...). Ils sont assimilés aux lignes d'une table dans une base de données.

Dans le diagramme d'entités précédent et grâce à l'analyse faite, il est simple de définir les classes d'Entity Beans. Nous retrouvons ainsi : `Resource`, `User`, `Category`, `Type`.

Utiliser les Session Beans

Alors que les Entity Beans représentent des « choses », les Session Beans effectuent le travail métier. Pour les identifier, une première étape consiste à regarder les fonctionnalités, et de les regrouper de façon logique (fig. 11.7).

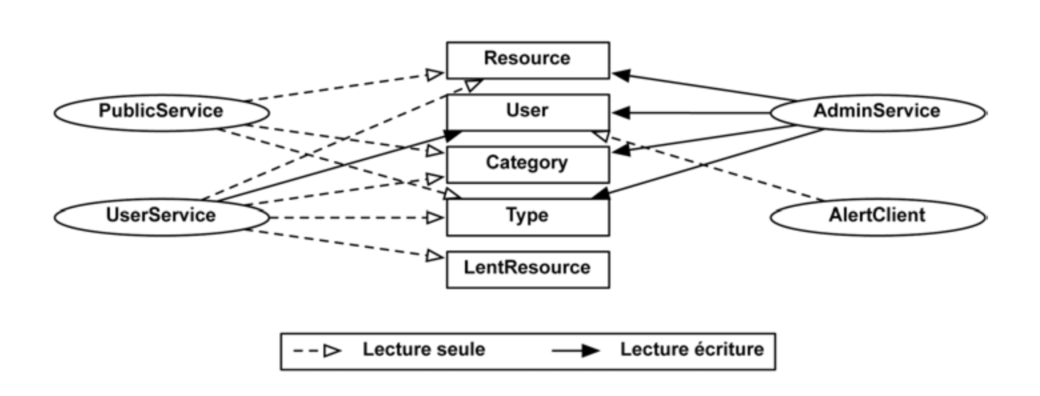


Figure 11.7 — Accès des différents services

Dans notre cas pratique, les objets suivants sont de bons candidats pour être des Session Beans :

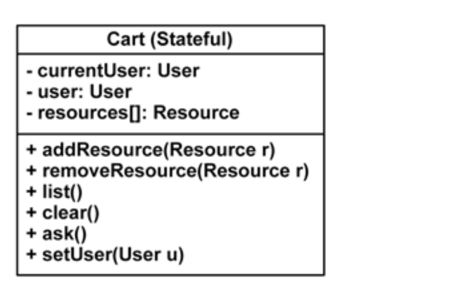
- `AdminService` gère le « back-office[1] » de l'application, c'est-à-dire qu'il permet à un utilisateur ayant les droits (administrateur) d'ajouter, modifier, supprimer les `Type, Category, User, Resource`.
- `PublicService` offre un accès en lecture seule aux informations de la base de données. Il est utilisé principalement pour rechercher et lister les ressources disponibles.
- `UserService` gère le « back-office » lié aux utilisateurs. Il offre la possibilité d'ajouter, de modifier et de supprimer les ressources de l'utilisateur. Celui-ci peut également visualiser ses ressources en location ou louées et leur statut.
- `AlertClientService` offre un accès en lecture seule aux informations concernant l'utilisateur connecté (profil, historique...).
- `Cart` gère les opérations liées au panier virtuel d'un utilisateur. Le principe de ce panier est d'offrir la possibilité à l'utilisateur de choisir un ensemble de ressources à louer puis de valider leur location d'un seul coup. C'est exactement le même principe que sur un site de commerce électronique.

Les Session Beans sont généralement utilisés pour travailler sur les données représentées par les Entity Beans. Ils représentent les actions associées à ces Entity Beans. Un Session Bean permet d'implémenter directement le *design pattern* « façade ». En effet, il cache, au client, l'accès à différents sous-systèmes (autre Session Bean, base de données, service de mail...).

Suivant la complexité d'une application, il est possible que plusieurs Session Beans implémentent une même méthode. Nous retrouvons ce cas-ci dans notre application avec la recherche de ressources. Cette méthode se retrouve dans `PublicService`, `AdminService` et `UserService`. Il est alors intéressant de créer un Session Bean commun regroupant ces différentes méthodes. Il est également possible de combiner les Session Beans entre eux si le développeur ne souhaite pas augmenter leur nombre.

Deux autres points sont à prendre en considération concernant les Session Beans. Nous avons vu qu'il en existait deux types : Stateless et Stateful. La différence entre ces deux types se situe au niveau de la sauvegarde de l'état conversationnel entre le client et le Session Bean. Avec le type Stateful, il est possible de partager des informations entre plusieurs appels de méthodes. Par exemple, l'objet `Cart` est lié à un utilisateur et contient un ensemble d'objets que celui-ci aimerait louer. Il semble impératif d'enregistrer l'état de celui-ci étant donné que l'utilisateur ajoute les éléments un à un dans son panier.

1. *Back-office* : partie d'un site web permettant d'en gérer le contenu de manière dynamique (ajoute/modifier/supprimer les informations, les utilisateurs...). Généralement accessible par authentification.



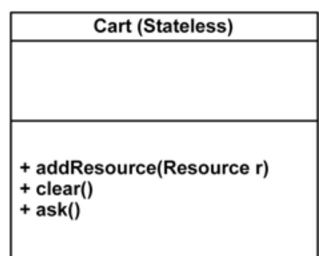


Figure 11.8 — Cart, Stateless ou Stateful ?

Certains diront qu'il existe pourtant une solution utilisant un Stateless Session Bean. Cependant cette solution pose certains problèmes : l'application cliente doit passer les paramètres liés à la conversation à chaque appel de méthodes. Parallèlement, on peut assimiler les Stateless Session Beans à des classes outils (ayant des méthodes statiques). Dès qu'un état conversationnel existe entre l'application et le Session Bean alors il est préférable d'utiliser le type Stateful.

Cependant ce dernier type a souvent été dénigré à cause des problèmes de performances qu'il peut engendrer. En effet, une instance du composant est créée à chaque demande du client (appel JNDI *via* `Context.lookup()`). Toutefois, une application bien conçue doit gérer la vie de ce composant en fonction des événements de l'application. Dans notre cas, une fois la commande de location d'objet réalisée, le panier n'est plus nécessaire (on peut alors demander la suppression de l'instance du Stateful Session Bean). L'autre point à prendre en compte est la mise en *cluster* de l'application. En effet, l'utilisation de Stateful Session Beans oblige le *cluster* à répliquer l'état des instances dans l'ensemble des nœuds du *cluster*. Cela peut créer du trafic réseau et impose du travail en plus pour les serveurs.

Choisir la visibilité des Session Beans

Le choix concernant l'utilisation d'une interface distante ou locale dépend réellement des besoins de l'application. La règle principale pour ce choix est : « Ne pas utiliser d'interface distante sauf si c'est réellement nécessaire ». En effet, l'utilisation d'un accès distant implique des complications et des pertes de performances qui ne sont pas toujours nécessaires. De plus, il est très souvent difficile de changer la structure d'une interface distante car cela implique généralement des modifications au niveau de l'application cliente (qui devra alors être redéployée…).

Dans la majorité des cas, les Session Beans sont appelés à l'intérieur du serveur d'applications (par l'application web, par exemple) et n'ont donc besoin que d'un accès local.

Dans notre cas pratique, seul le Session Bean `AlertClientService` propose un accès distant. Il est utilisé par l'application distante Java et offre l'accès au récapitulatif du compte (objets à rendre, objets en cours de prêt…). Cela impose que les

objets retournés par ce composant puissent être transmis à travers le réseau. Il faut donc pour cela, qu'ils soient sérialisables (implémentant l'interface `Serializable` ou `Externalizable`).

11.3 PASSONS À LA PRATIQUE

Dans cette partie, nous passons à la concrétisation du contrat « louer des objets ». Celui-ci regroupe les cas d'utilisation suivants :

- Inscription,
- Authentification,
- Recherche d'un objet,
- Ajout d'un objet au panier de locations,
- Validation du panier de locations.

Nous accentuerons nos explications sur la couche métier qui est réalisée avec la technologie EJB 3.

Nous avons choisi d'utiliser JSF pour la couche présentation. Cependant, nous ne nous étendrons pas sur l'explication des codes JSF, cette technologie n'étant pas le sujet de cet ouvrage.

11.3.1 Préparation de l'environnement

Pré-requis

Pour être à même de mettre en place ce cas pratique, vous devrez installer :

- NetBeans 5.5.
- MySQL 4 ou supérieur comprenant une base de données et un compte utilisateur configuré.
- Glassfish 1.0 ou supérieur

Création des projets

Il faut, avant toute chose, préparer l'environnement de travail au sein de l'IDE. Nous utiliserons, ici, NetBeans pour cet exemple.

- Créez un nouveau projet de type Entreprise → Entreprise Application (fig. 11.9).

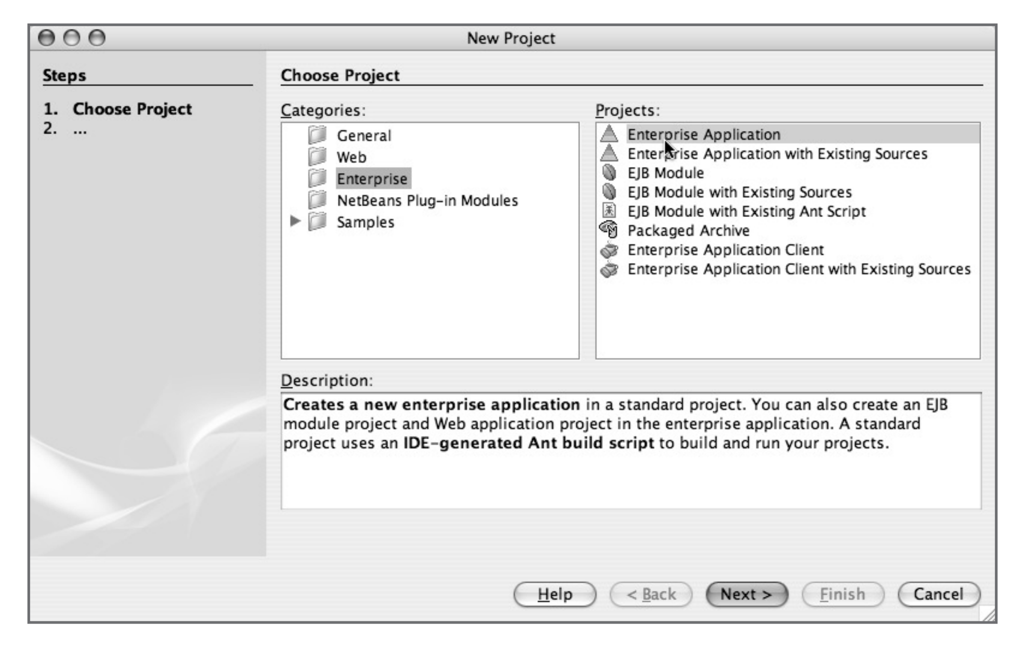


Figure 11.9 – Création du projet

- Paramétrez votre projet comme sur la figure 11.10 :

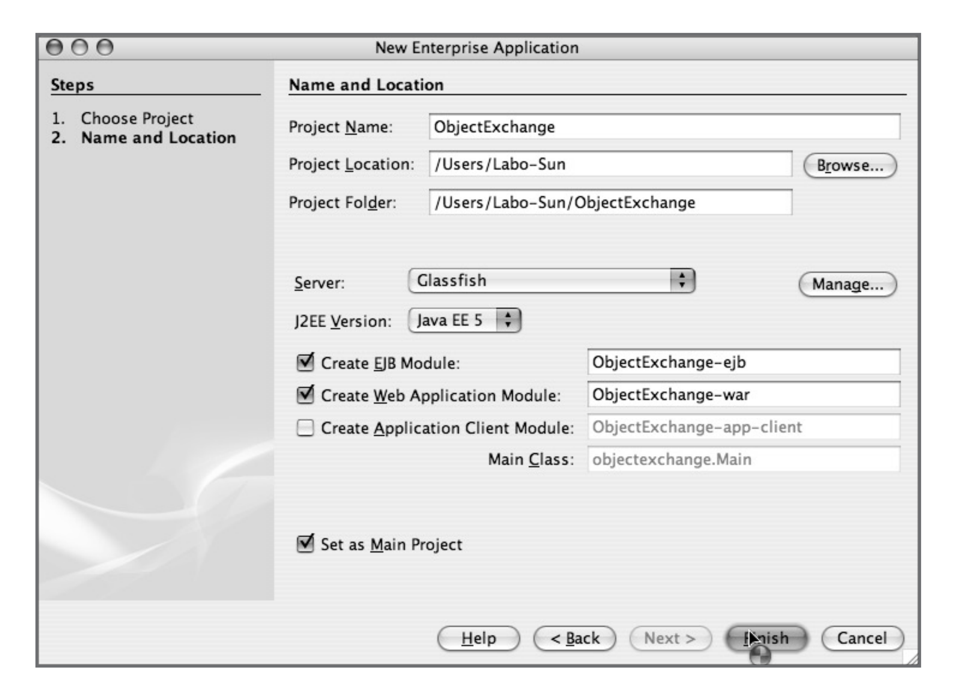


Figure 11.10 – Paramétrage du projet



- Pensez à ajouter votre serveur glassfish précédemment installé *via* le bouton « Manage ».
- Cliquez sur « Finish ».

La figure 11.11 présente l'arborescence du projet généré par NetBeans.



Figure 11.11 — Arborescence du projet

Le projet « ObjectExchange » représente le bloc principal de l'application. Vous remarquez la présence des projets « ObjectExchange-ejb » et « ObjectExchange-war » en tant que « Module J2EE » de ce projet (archives EAR).

Le projet « ObjectExchange-ejb » regroupe l'ensemble des composants EJB de l'application (archive JAR).

Le projet « ObjectExchange-war » regroupe les pages web et les composants JSF (archive WAR).

Ajout des librairies

Toutes les librairies liées aux spécifications Java EE sont automatiquement incluses par NetBeans. Elles se trouvent dans Glassfish.

Toutefois, il est nécessaire d'importer les librairies supplémentaires nous permettant d'utiliser les *Facelets* et des composants Apache pour la partie présentation.

Remarque : Vous pouvez télécharger les librairies des *Facelets* standard sur https://facelets.dev.java.net/. De même, les librairies Apache sont disponibles sur http://jakarta.apache.org/commons/. Les extensions JSF d'Apaches sont disponibles sur http://myfaces.apache.org/tomahawk/index.html.

Ajoutez les librairies suivantes dans le projet « ObjectExchange-war » :

- el-api.jar
- el-ri.jar
- jsf-facelets.jar
- commons-beanutils-1.7.0.jar
- commons-collections-3.1.jar
- commons-digester-1.6.jar
- commons-el-1.0.jar
- commons-fileupload-1.1.1.jar
- commons-io-1.2.jar
- commons-lang-2.1.jar
- commons-logging-1.0.4.jar
- tomahawk-1.1.3.jar

Pour cela, faites un clic droit sur le dossier « Libraries » de « ObjectExchange-war » puis cliquez sur « Add JAR/Folder ».

Les EJB étant utilisés au sein de l'application web, il faut lier celle-ci avec l'application EJB. Pour cela, faites un clic droit sur le dossier « Libraries » de « ObjectExchange-war », cliquez sur « Add Project » puis sélectionnez le projet « ObjectExchange-ejb » (fig. 11.12).

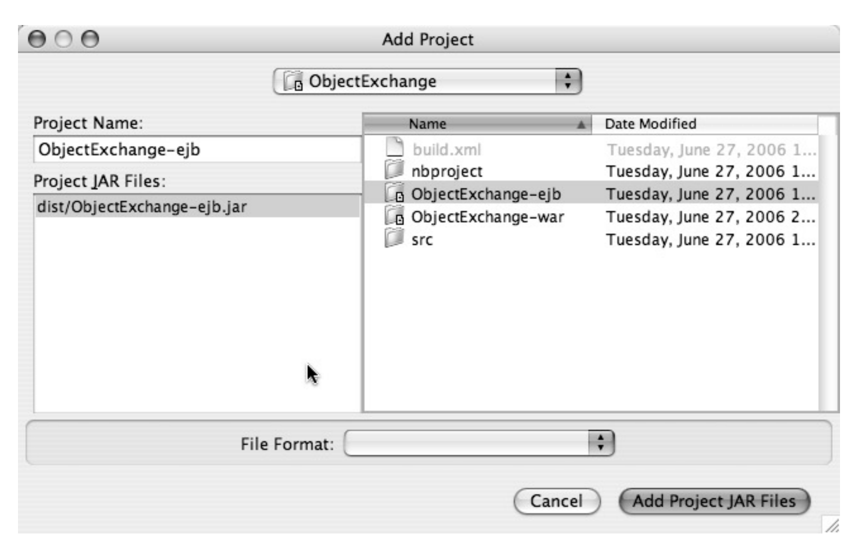


Figure 11.12 — Référence vers le projet EJB depuis le projet web

Création des fichiers de configuration

Nous devons intégrer les JSF, *Facelets* et les extensions à notre application web. Pour cela, nous devons modifier le fichier web.xml comme suit :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http:/
/java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <context-param>
    <param-name>com.sun.faces.verifyObjects</param-name>
    <param-value>false</param-value>
  </context-param>
  <context-param>
    <param-name>com.sun.faces.validateXml</param-name>
    <param-value>true</param-value>
  </context-param>
  <context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/faces-config.xml</param-value>
  </context-param>
  <context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
  </context-param>

  <!-- déclaration de la servlet JSF -->
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- mapping des pages JSF (suffixe jsf) -->
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Les balises `<context>` permettent de définir des paramètres associés au contexte de l'application. Ces paramètres sont, ici, utilisés par la servlet de JSF, déclarée *via* la balise `<servlet>`. L'ensemble des requêtes dont l'URL se termine par « jsf » seront traitées par JSF grâce au *mapping* réalisé *via* la balise `<servlet-mapping>`.

Pour finir, le fichier faces-config.xml contenant la configuration JSF doit être créé dans le dossier « Web Pages/WEB-INF » comme désigné par le paramètre du contexte `javax.faces.CONFIG_FILES` dans le fichier précédent (web.xml).

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config version="1.2" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http:/
```

```
/java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd">

  <application>
    <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
  </application>

</faces-config>
```

Comme nous utilisons le framework *Facelet*, nous devons spécifier à JSF que nous utilisons celui-ci pour la génération de la vue. Pour cela, il suffit d'utiliser la balise `<view-handler>` et de lui spécifier la classe à utiliser (ici : `com.sun.facelets.FaceletViewHandler`).

11.3.2 Test de l'environnement

Dans cette partie, nous mettons en place la structure minimale de l'application et la testons par une simple page d'accueil.

Création des fichiers de bases

L'application web utilise la technologie *Facelet* qui permet de gérer facilement des *templates*. Il faut donc créer ces modèles en premier. Nous allons les organiser comme le représente la figure 11.13.

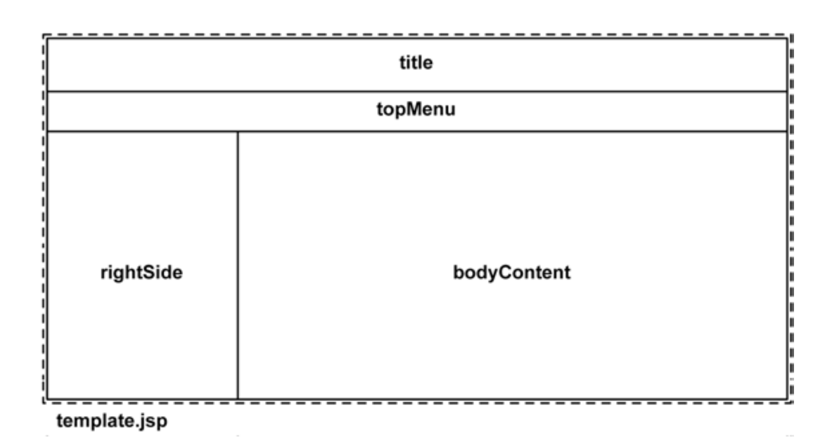


Figure 11.13 — Schéma template

Voici le fichier de base de *template* qui représente la base générale de l'interface web :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
            "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
<head>
```

```
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
    <title><ui:insert name="title">ObjectExchange</ui:insert></title>
    <link href="css/screen.css" rel="stylesheet" type="text/css" />
  </head>
  <body id="mainBody">
    <div id="page">
      <div id="header"><span>
        <ui:insert name="title">ObjectExchange</ui:insert>
      </span></div>
      <div id="nav">
        <ui:insert name="top">
          <ul>
            <li id="current">Menu1</li>
            <li><a href="#">Menu2</a></li>
            <li><a href="#">Menu3</a></li>
          </ul>
        </ui:insert>
      </div>
      <div id="container">
        <div id="leftside">
          <ui:insert name="leftside">
            <div class="box">
              <dl>
                <dt class="boxHeader">Titre :</dt>
                <dd class="boxContent">
                  <dl>
                    <dd>ligne1</dd>
                    <dd>ligne2</dd>
                  </dl>
                </dd>
              </dl>
            </div>
          </ui:insert>
        </div>
        <div id="content">
          <ui:insert name="bodycontent">
            Contenu de la page
          </ui:insert>
        </div>
      </div>
    </div>
  </body>
  </html>
```

Remarque : n'oubliez pas de créer votre fichier CSS et de préciser dans template.jsp pour qu'il soit pris en compte dans toute l'application.

Nous ne rentrons pas dans les détails du code XHTML, cependant vous pouvez remarquer que les balises `<ui:insert>` insert le contenu de l'attribut passé en paramètre (attribut `name`).

Par exemple, `<ui:insert name="title"> ObjectExchange </ui:insert>` insert le contenu désigné par la clé `title`, dans le cas où aucune clé n'est trouvée, le corps de la balise est exécutée et/ou affiché (ici `ObjectExchange`).

La partie publique de cette application est définie par le fichier template_public.jsp qui regroupe les différents éléments nécessaires à tout visiteur (login, inscription, panier...) et hérite de template.jsp.

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:s="http://jboss.com/products/seam/taglib"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf"
  xmlns:h="http://java.sun.com/jsf/html">
  <body>
    <ui:composition template="/WEB-INF/template/template.jsp">
      <ui:define name="top">
        <ui:include src="/WEB-INF/incl/menu_public.jsp"/>
      </ui:define>

      <ui:define name="leftside">
        <ui:include src="/WEB-INF/incl/search_resource.jsp" />
        <ui:include src="/WEB-INF/incl/login.jsp" />
        <ui:include src="/WEB-INF/incl/little_cart.jsp" />
      </ui:define>

      <ui:define name="bodycontent">
         <ui:insert name="body" />
      </ui:define>

    </ui:composition>
  </body>
</html>
```

La balise `<ui:composition>` permet de définir que l'on souhaite utiliser une structure définie dans un autre fichier et préciser différents éléments *via* `<ui:define>`. Cette dernière est composée de la balise `<ui:include>` permettant d'inclure différents fichiers JSP. Ces fichiers étant nécessaires pour notre premier test d'exécution de l'application, vous pouvez dès maintenant les créer avec le contenu temporaire suivant.

```
<div xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:c="http://java.sun.com/jstl/core" class="box" >

</div>
```

Ce code génère simplement le code HTML suivant : `<div class="box"></div>`.

Création de la page d'accueil

Afin de vérifier que l'ensemble des étapes de paramétrages de l'environnement et de l'application fonctionne, nous devons créer une page de test. Pour cela, créez un fichier `public_index.jsp` dans le dossier « Web Pages » de votre projet « ObjectExchange-war » dont le contenu est mentionné ci-après.

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:s="http://jboss.com/products/seam/taglib"
  xmlns:ui="http://java.sun.com/jsf/facelets"
```

```
  xmlns:f="http://java.sun.com/jsf"
  xmlns:h="http://java.sun.com/jsf/html">
  <body>
  <ui:composition template="/WEB-INF/template/template_public.jsp">
      <ui:param name="page" value="public_home" />
      <ui:define name="body">
        <h1>Bienvenue sur ObjectExchange</h1>
        <p>
          Bienvenue sur ObjectExchange où vous pouvez
          partager vos objets
          <br/>
        </p>
      </ui:define>
    </ui:composition>
  </body>
</html>
```

Vous pouvez remarquer l'utilisation de la composition `template_public.jsp` précédemment créée. La balise `<ui:param>` définit un paramètre `page`. Celui-ci pourra être utilisé en aval pour connaître la page en cours, afin d'en modifier le menu en conséquence, par exemple (affichage différent par rapport au menu courant).

Pour pouvoir tester l'application, il faut impérativement que notre projet EJB possède un composant (Entity Bean, Session Bean ou Message Driven Bean). Nous allons donc créer notre premier Session Bean. Pour cela, faites un clic droit sur le projet « ObjectExchange-ejb » puis sur « New » et enfin sur « Session Bean » (fig. 11.14).

Figure 11.14 – Création du Session Bean PublicServiceBean

Voici le code de l'interface métier PublicService :

```
@Local
public interface PublicService {
}
```

Voici maintenant le code de la classe d'implémentation PublicServiceBean :

```
@Stateless
public class PublicServiceBean implements PublicService {
}
```

Pour le moment, nous ne définissons aucune méthode métier. Celles-ci seront ajoutées et expliquées plus loin.

Lancement de l'application

Maintenant que la structure de notre application est complète, nous pouvons passer au test.

Pour lancer l'application, vous pouvez utiliser la touche F6 ou le menu « Run » puis « Run Main Project » (fig. 11.15).

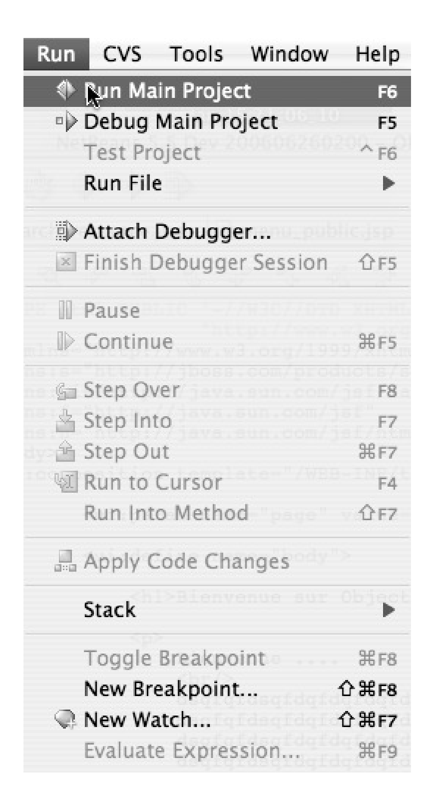


Figure 11.15 — Lancement du projet

NetBeans lance automatiquement le navigateur web à l'adresse de base de l'application. Pour voir la page d'accueil, il vous suffit d'aller à l'adresse suivante : http://localhost:8080/ObjectExchange-war/public_index.jsf.

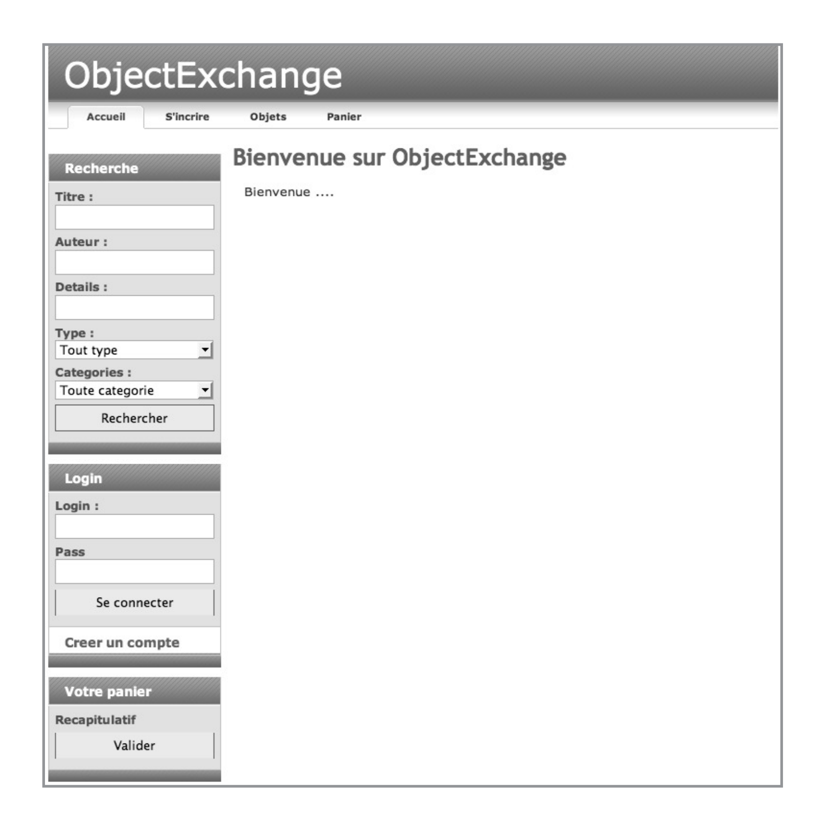


Figure 11.16 — Page d'accueil de l'application

La page public_index.jsf est automatiquement associée, par JSF, au fichier public_index.jsp créé précédemment (fig. 11.16).

11.3.3 Développement de la couche métier

Création des Entity Beans

L'étape de développement des Entity Beans est cruciale dans une application. Elle représente le résultat de l'analyse du besoin au niveau des données devant être gérées par l'application.

Pour ajouter un nouvel Entity Bean, faites un clic droit sur le projet « ObjectExchange-ejb » puis « New » et enfin « Entity Class » (fig. 11.17).

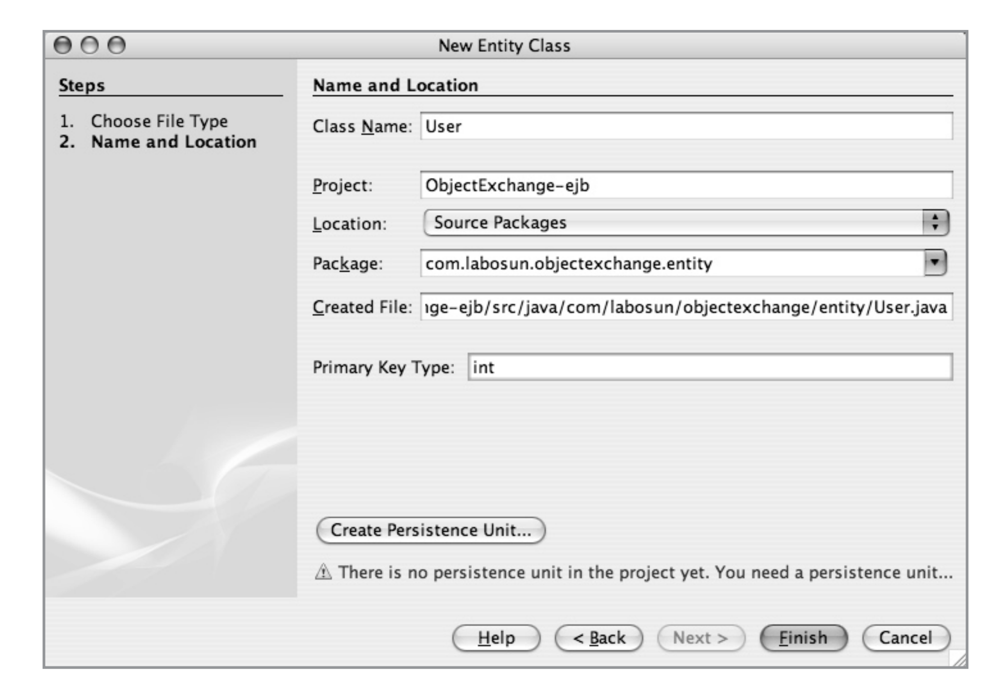


Figure 11.17 – Création du premier Entity Bean

Un message vous informe qu'aucune unité de persistance n'existe (voir chapitre 6). Vous devez alors en créer une *via* le bouton « Create Unit Persistent » (fig. 11.18).

Remarque : le message n'apparaît plus une fois le fichier persistence.xml créé.

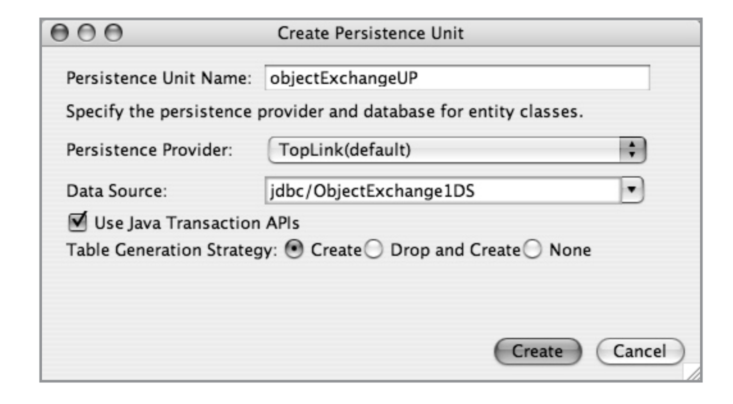


Figure 11.18 – Création de l'unité de persistance (persistence.xml)

À la fin de cet assistant, le fichier persistence.xml est automatiquement créé et se trouve dans le dossier « Configuration Files ».

Ajoutez ensuite l'ensemble des propriétés adéquates à cette entité. Puis reproduisez ces opérations pour les autres entités (`Category`, `Type`, `Resource`, `LentResource`). Référez-vous au diagramme d'entité présenté précédemment dans la partie analyse (fig. 11.6) pour connaître les différentes relations entre celles-ci.

> **Conseil :** Vous pouvez générer les *getters/setters* en faisant un clic droit sur votre fichier, puis « Tool », « Refractor » et « Encapsulate Fields ».

Voici l'Entity Bean `User` :

```
@Entity
@Table(name="NetUser")
@NamedQueries( {
  @NamedQuery(name = "findAllUsers", query = "SELECT u FROM User u"),
  @NamedQuery(name = "findUserById", query = "SELECT u FROM User u WHERE u.id =
:id"),
  @NamedQuery(name = "findUserByEmail", query = "SELECT u FROM User u WHERE
u.email = :email"),
  @NamedQuery(name = "identifyUser", query = "SELECT u FROM User u WHERE u.email
= :email AND u.password = :password")
})
public class User implements Serializable {

  private int id;
  private String firstName;
  private String lastName;
  private String email;
  private String password;
  private String address;
  private String zipCode;
  private String city;
  private String country;
  private boolean active;
  private int level;

  private Collection<Resource> resources;
  private Collection<LentResource> lentResources;

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstname) {
    this.firstName = firstname;
  }
```

```
public String getLastName() {
  return lastName;
}

public void setLastName(String lastname) {
  this.lastName = lastname;
}

public String getEmail() {
  return email;
}

 public void setEmail(String email) {
  this.email = email;
}

public String getPassword() {
  return password;
}

public void setPassword(String password) {
  this.password = password;
}

public String getAddress() {
  return address;
}

public void setAddress(String address) {
  this.address = address;
}

public String getZipCode() {
  return zipCode;
}

public void setZipCode(String zipCode) {
  this.zipCode = zipCode;
}

public String getCity() {
  return city;
}

public void setCity(String city) {
  this.city = city;
}

public String getCountry() {
  return country;
}

public void setCountry(String country) {
  this.country = country;
}
```

```
  public boolean isActive() {
    return active;
  }

  public void setActive(boolean active) {
    this.active = active;
  }

  public int getLevel() {
    return level;
  }

  public void setLevel(int level) {
    this.level = level;
  }

  public int hashCode() {
    int hash = 0;
    Integer id = Integer.valueOf(this.getId());
    hash += (id != null ? id.hashCode() : 0);
    return hash;
  }

  public boolean equals(Object object) {
    Integer id = Integer.valueOf(this.getId());
    if (object == null || !this.getClass().equals(object.getClass())) {
      return false;
    }
    User other = (User)object;
    if (this.getId() != other.getId() && (id == null ||
!id.equals(other.getId()))) return false;
    return true;
  }

  public String toString() {
    return "" + this.getFirstName();
  }
}
```

Voici l'Entity Bean `Category` :

```
@Entity
@NamedQueries({
  @NamedQuery(name = "findAllCategories", query = "SELECT c FROM Category c"),
  @NamedQuery(name = "findCategoryById", query = "SELECT c FROM Category c WHERE
c.id = :id")}
)
public class Category implements Serializable{

  private int id;
  private String label;

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  public int getId() {
    return id;
  }
```

```
  public void setId(int id) {
    this.id = id;
  }

  public String getLabel() {
    return label;
  }

  public void setLabel(String label) {
    this.label = label;
  }

  public int hashCode() {
    int hash = 0;
    Integer id = Integer.valueOf(this.id);
    hash += (id != null ? id.hashCode() : 0);
    return hash;
  }

  public boolean equals(Object object) {
    Integer id = Integer.valueOf(this.id);
    if (object == null || !this.getClass().equals(object.getClass())) {
      return false;
    }
    Category other = (Category)object;
    if (this.id != other.id && (id == null || !id.equals(other.id))) return
false;
    return true;
  }

  public String toString() {
    return "" + this.label;
  }
}
```

Voici l'Entity Bean Type :

```
@Entity
@NamedQueries( {
  @NamedQuery(name = "findAllTypes", query = "SELECT type FROM Type AS type"),
  @NamedQuery(name = "findTypeById", query = "SELECT t FROM Type t WHERE t.id =
:id")}
)
public class Type implements Serializable{
  private int id;
  private String label;

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }
```

```
  public String getLabel() {
    return label;
  }

  public void setLabel(String label) {
    this.label = label;
  }

  public int hashCode() {
    int hash = 0;
    Integer id = Integer.valueOf(this.id);
    hash += (id != null ? id.hashCode() : 0);
    return hash;
  }

  public boolean equals(Object object) {
    Integer id = Integer.valueOf(this.id);
    if (object == null || !this.getClass().equals(object.getClass())) {
      return false;
    }
    Type other = (Type)object;
    if (this.id != other.id && (id == null || !id.equals(other.id))) return
false;
    return true;
  }

  public String toString() {
    return "" + this.label;
  }
}
```

Voici l'Entity Bean `Resource` :

```
@Entity
@NamedQueries( {
  @NamedQuery(name = "findAllResources", query = "SELECT r FROM Resource r ORDER
BY r.date"),
  @NamedQuery(name = "findAllAvailableResources", query = "SELECT r FROM
Resource r WHERE r.valid = true AND r.active = true AND 0 = " +
        "(SELECT COUNT(lent) " +
        "FROM LentResource lent " +
        "WHERE lent.resource = r AND NOT (lent.state = :state)) ORDER BY
r.date"),
  @NamedQuery(name = "findResourceById", query = "SELECT r FROM Resource r WHERE
r.id = :id"),
  @NamedQuery(name = "findResourceByUserIdAndResourceId", query = "SELECT r FROM
Resource r WHERE r.id = :resourceId AND r.user.id = :userId"),
  @NamedQuery(name = "findAvailableResourceById", query = "SELECT r FROM
Resource r WHERE r.id = :id AND r.valid = true AND r.active = true AND 0 = " +
        "(SELECT COUNT(lent) " +
        "FROM LentResource lent " +
        "WHERE lent.resource = r AND NOT (lent.state = :state))"),
  @NamedQuery(name = "findResourceByUserId", query = "SELECT r FROM Resource r
WHERE r.user.id = :userId")
})
public class Resource implements Serializable {
```

```
private int id;

private String author;
private String title;
private Date date;
private int defaulftLoanDuration;
private boolean valid;
private boolean active;
private byte[] picture;
private String details;

private Type type;
private User user;
private Set<Category> categories;

public Resource() {
  type = new Type();
  user = new User();
  categories = new HashSet<Category>();
}

@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
public int getId() {
  return id;
}

public void setId(int id) {
  this.id = id;
}

public String getAuthor() {
  return author;
}

public void setAuthor(String author) {
  this.author = author;
}

public String getTitle() {
  return title;
}

public void setTitle(String title) {
  this.title = title;
}

public String getDetails() {
  return details;
}

public void setDetails(String details) {
  this.details = details;
}

@Lob
```

```
@Basic(fetch=FetchType.LAZY)
public byte[] getPicture() {
  return picture;
}

public void setPicture(byte[] picture) {
  this.picture = picture;
}

public boolean getActive() {
  return active;
}

public void setActive(boolean active) {
  this.active = active;
}

public boolean getValid() {
  return valid;
}

public void setValid(boolean valid) {
  this.valid = valid;
}

@Temporal(TemporalType.DATE)
public Date getDate() {
  return date;
}

public void setDate(Date date) {
  this.date = date;
}

public int getDefaulftLoanDuration() {
  return defaulftLoanDuration;
}

public void setDefaulftLoanDuration(int defaulftLoanDuration) {
  this.defaulftLoanDuration = defaulftLoanDuration;
}

@ManyToOne
@JoinColumn(name = "user_fk")
public User getUser() {
  return user;
}

public void setUser(User user) {
  this.user = user;
}

@ManyToOne
@JoinColumn(name="type_fk")
public Type getType() {
  return type;
}
```

```
  public void setType(Type type) {
    this.type = type;
  }

  @ManyToMany
  @JoinTable(
  name="resources_categories",
      joinColumns=
      @JoinColumn(name="RESOURCE_ID", referencedColumnName="ID"),
  inverseJoinColumns=
      @JoinColumn(name="CATEGORY_ID", referencedColumnName="ID")
      )
  public Set<Category> getCategories() {
    return categories;
  }

  public void setCategories(Set<Category> categories) {
    this.categories = categories;
  }

  public int hashCode() {
    int hash = 0;
    Integer id = Integer.valueOf(this.id);
    hash += (id != null ? id.hashCode() : 0);
    return hash;
  }

  public boolean equals(Object object) {
    Integer id = Integer.valueOf(this.id);
    if (object == null || !this.getClass().equals(object.getClass())) {
      return false;
    }
    Resource other = (Resource)object;
    if (this.id != other.id && (id == null || !id.equals(other.id))) return
false;
    return true;
  }

  public String toString() {
    return "" + this.title;
  }
}
```

L'Entity Bean Resource est plus intéressant en termes de code. En effet, ce composant possède différentes relations :

- « Many To Many » avec Category.
- « Many To One » avec Type.
- « Many To One » avec User.

De plus, la propriété picture est annotée avec @Lob afin qu'elle soit prise en compte comme un BLOB dans la base de données. Nous spécifions qu'elle doit être chargée à la demande car nous n'accédons pas systématiquement à sa valeur pour

chaque ressource chargée. Cela a l'avantage d'éviter les accès base de données inutiles mais à l'inconvénient d'obliger au développeur d'initialiser cette propriété lorsque l'application cliente l'utilise.

Voici l'Entity Bean `LentResource` :

```
@Entity
@NamedQueries( {
  @NamedQuery(name = "findAllLentResources", query = "SELECT l FROM LentResource
l"),
    @NamedQuery(name = "findAllRentResourcesByUserId", query = "SELECT l FROM
LentResource l WHERE l.user.id = :userId"),
    @NamedQuery(name = "findAllLentResourcesByUserId", query = "SELECT l FROM
LentResource l WHERE l.resource.user.id = :userId"),
  @NamedQuery(name = "findLentResourceById", query = "SELECT l FROM LentResource
l WHERE l.id = :id")}
)
public class LentResource implements Serializable {

  public enum StateType {
    LOAN_WAITING, LENT, RETURN_WAITING, RETURNED
  };

  private int id;
  private Date loanStartDate;
  private Date loanEndDate;

  private StateType state;

  private Resource resource;
  private User user;


  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  @Temporal(TemporalType.DATE)
  public Date getLoanStartDate() {
    return loanStartDate;
  }

  public void setLoanStartDate(Date loanStartDate) {
    this.loanStartDate = loanStartDate;
  }

  @Temporal(TemporalType.DATE)
  public Date getLoanEndDate() {
    return loanEndDate;
  }
```

```
    public void setLoanEndDate(Date loanEndDate) {
      this.loanEndDate = loanEndDate;
    }


    @Enumerated(value = EnumType.STRING)
    // EnumType.ORDINAL est utilisé par défaut
    @Column(length = 15, nullable=false)
    public StateType getState() {
      return state;
    }

    public void setState(StateType state) {
      this.state = state;
    }

      @Transient
      public boolean isLoanWaitingState() {
        return StateType.LOAN_WAITING.equals(state);
      }

      @Transient
      public boolean isLentState() {
        return StateType.LENT.equals(state);
      }

      @Transient
      public boolean isReturnedState() {
        return StateType.RETURNED.equals(state);
      }

      @Transient
      public boolean isReturnWaitingState() {
        Date now = new Date();
        try {
          return ((!StateType.RETURNED.equals(state)) && now.getTime() <
  getLoanEndDate().getTime());
        }
        catch(Exception e) {
          return false;
        }
      }

    @ManyToOne
    @JoinColumn(name = "resource_fk")
    public Resource getResource() {
      return resource;
    }

    public void setResource(Resource resource) {
      this.resource = resource;
    }

    @ManyToOne
    @JoinColumn(name = "user_fk")
    public User getUser() {
```

```
      return user;
    }

    public void setUser(User user) {
      this.user = user;
    }

    public int hashCode() {
      int hash = 0;
      Integer id = Integer.valueOf(this.id);
      hash += (id != null ? id.hashCode() : 0);
      return hash;
    }

    public boolean equals(Object object) {
      Integer id = Integer.valueOf(this.id);
      if (object == null || !this.getClass().equals(object.getClass())) {
        return false;
      }
      LentResource other = (LentResource)object;
      if (this.id != other.id && (id == null || !id.equals(other.id))) return
  false;
      return true;
    }

    public String toString() {
      return "" + this.id;
    }
  }
```

Cet Entity Bean utilise également de nombreux concepts vus dans cet ouvrage. Tout d'abord, les états d'une location (ou d'un prêt suivant la personne qui regarde la valeur de cet objet) sont définis par l'énumération `StateType`. Comme nous souhaitons enregistrer cet état sous forme de chaîne de caractère dans la base de données, nous devons le spécifier au conteneur *via* l'annotation `@Enumerated`.

Ensuite, ce composant implémente quelques méthodes métiers simples le concernant. Nous pouvons facilement vérifier l'état de cette location par les méthodes `isLoanWaitingState`, `isLentState`, `isReturnedState` ou encore `isReturnWaitingState`. Celles-ci étant considérées comme des *getters*, nous devons spécifier au conteneur de ne pas les enregistrer en base de données par l'intermédiaire de l'annotation `@Transient`.

Création des Session Beans

Les Session Beans sont créés pour effectuer les traitements métiers et gérer les données. Nous avons besoin de deux Session Beans pour gérer le processus complet de location :

- `PublicService` : gère les méthodes métiers génériques (inscription, recherche) accessibles à tout visiteur.
- `CartService` : gère le panier virtuel (ajout, suppression et validation).

Voici le détail de `PublicServiceBean` :

```
@Stateless
public class PublicServiceBean implements PublicService {

  @PersistenceContext(unitName="objectExchangeUP")
  private EntityManager em;

  @javax.annotation.Resource(name="passwordLength")
  private int passwordLength;

  @javax.annotation.Resource(name="nbLastResource")
  private int nbLastResource;

  public PublicServiceBean() {
  }

  public User join(User user) throws UserAlreadyJoinedException {
    // vérifie qu'un utilisateur n'existe pas avec le même login ou email
    List<User> results =
em.createNamedQuery("findUserByEmail").setParameter("email",
user.getEmail()).getResultList();
    if(results.size() == 0) {
      // enregistre l'utilisateur
      user.setPassword(StringUtil.createRandomString(passwordLength));
      em.persist(user);
      return user;
    } else {
      throw new UserAlreadyJoinedException();
    }
  }

  public User login(String email, String password) throws UserNotLoggedException
{
    try {
      User logged = (User)
em.createNamedQuery("identifyUser").setParameter("email",email).setParameter("pa
ssword", password).getSingleResult();
      return logged;
    } catch(Exception e) {
      throw new UserNotLoggedException(e);
    }
  }

  public List<Resource> searchResource(Resource resourceCriteria) {
    HashMap<String, Object> conditions = new HashMap<String, Object>();

    // titre
    if(resourceCriteria.getTitle() != null) {
      conditions.put("title", resourceCriteria.getTitle());
    }
    // auteur
    if(resourceCriteria.getAuthor() != null) {
      conditions.put("author", resourceCriteria.getAuthor());
    }
    // détail
    if(resourceCriteria.getDetails() != null) {
```

```
      conditions.put("details", resourceCriteria.getDetails());
    }
    // type
    if(resourceCriteria.getType() != null) {
      conditions.put("type", resourceCriteria.getType());
    }

    // génération d'une requête personnalisée suivant
    // les valeurs de la ressource
    StringBuffer searchQL = new  StringBuffer("SELECT res " +
        "FROM Resource res, IN(res.categories) rescat " +
        "WHERE res.valid = true AND res.active = true AND 0 = " +
        "(SELECT COUNT(lent) " +
        "FROM LentResource lent " +
        "WHERE lent.resource = res AND NOT (lent.state = :state))");
    Iterator<String> keysIt = conditions.keySet().iterator();
    String condition;
    while(keysIt.hasNext()) {
      condition = keysIt.next();
      String conditionalOperator = (conditions.get(condition) instanceof String)
? "LIKE" : "=";
      searchQL.append(" AND res." + condition + " " + conditionalOperator + " :"
+ condition);
    }

    // catégorie
    if(resourceCriteria.getCategories().size() > 0) {
      searchQL.append(" AND rescat = :category ");
    }

    // création de l'objet Query associé
    Query searchQuery = em.createQuery(searchQL.toString());

    // assignation des paramètres
    keysIt = conditions.keySet().iterator();
    Object conditionValue ;
    while(keysIt.hasNext()) {
      condition = keysIt.next();
      conditionValue = conditions.get(condition);
      if(conditionValue instanceof String) {
        conditionValue = "%" + conditionValue + "%";
      }
      searchQuery.setParameter(condition, conditionValue);
    }

    // état des locations
    searchQuery.setParameter("state", LentResource.StateType.RETURNED);

    // catégorie
    if(resourceCriteria.getCategories().size() > 0) {
      searchQuery.setParameter("category",
resourceCriteria.getCategories().iterator().next());
    }

    // exécute la requête
    return searchQuery.getResultList();
  }
```

```
  public List<Resource> findLastResource() {
    Query q = em.createNamedQuery("findAllAvailableResources");
    q.setMaxResults(nbLastResource);
    q.setParameter("state", LentResource.StateType.RETURNED);
    return q.getResultList();
  }

  public Resource findResource(int resourceId) {
    Resource res = em.find(Resource.class, resourceId);
    // initialise les catégories
    res.getCategories().size();
    return res;
  }

  public byte[] getResourcePicture(int resourceId) {
    Query query = em.createNamedQuery("findResourceById")
      .setParameter("id", resourceId);
    Resource r = (Resource) query.getSingleResult();
    return r.getPicture();
  }

  public Collection<Category> findAllCategory() {
    return em.createNamedQuery("findAllCategories").getResultList();
  }

  public Collection<Type> findAllType() {
    return em.createNamedQuery("findAllTypes").getResultList();
  }
}
```

Ce Session Bean est de type `@Stateless`. En effet, ses méthodes métiers sont génériques et n'ont pas besoin de garder d'état conversationnel avec l'application cliente l'utilisant. Remarquez l'appel à la méthode `size()` sur la liste des catégories liées à la ressource chargée dans la méthode `findResource()`.

```
  res.getCategories().size();
```

Cet appel est obligatoire si l'application cliente souhaite récupérer la liste des catégories. Si aucune initialisation n'est faite, une exception sera levée. Cet appel demande donc au moteur de persistance de charger la liste des catégories associée à la ressource courante.

Nous demandons au conteneur l'injection de différents éléments lors de la création du composant :

- Le premier élément est l'unité de persistance permettant de travailler avec les données *via* `@PersistenceContext` (voir chapitre 6).
- Les seconds éléments représentent des variables d'environnement déclarées dans le fichier de déploiement EJB « ejb-jar.xml ». Ce fichier doit être créé et placé dans le dossier « Configuration Files ». Voici le contenu de celui-ci :

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http:/
/java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">

  <enterprise-beans>
    <session>
      <ejb-name>
        PublicServiceBean
       </ejb-name>

       <env-entry>
         <env-entry-name>passwordLength</env-entry-name>
         <env-entry-type>java.lang.Integer</env-entry-type>
         <env-entry-value>8</env-entry-value>
       </env-entry>

       <env-entry>
         <env-entry-name>nbLastResource</env-entry-name>
         <env-entry-type>java.lang.Integer</env-entry-type>
         <env-entry-value>10</env-entry-value>
       </env-entry>

    </session>
  </enterprise-beans>

</ejb-jar>
```

Nous déclarons dans ce fichier, les entrées `passwordLength` et `nbLastResource` utilisées dans le Session Bean précédent.

Voici le détail de `CartServiceBean` :

```
@Stateful
@Local( { CartService.class })
public class CartServiceBean implements CartService {

  @PersistenceContext(unitName = "objectExchangeUP")
  private EntityManager objectExchangeEM;

  private Collection<Resource> resources;

  public CartServiceBean() {
    resources = new ArrayList<Resource>();
  }

  public void add(int resourceId) {
    System.out.println("add : " + resourceId);
    Iterator<Resource> it = resources.iterator();
    while(it.hasNext()) {
      Resource res = it.next();
      if(res.getId() == resourceId) {
        return;
      }
    }
```

```
      Query q = objectExchangeEM.createNamedQuery("findAvailableResourceById");
      q.setParameter("state", LentResource.StateType.RETURNED);
      q.setParameter("id", resourceId);
      Resource resourceToAdd = (Resource) q.getSingleResult();
      if(resourceToAdd != null)
        resources.add(resourceToAdd);
    }

    public void remove(int resourceId) {
      Iterator<Resource> it = resources.iterator();
      while(it.hasNext()) {
        Resource res = it.next();
        if(res.getId() == resourceId) {
          it.remove();
          return;
        }
      }
    }

    public void resetCart() {
      resources.clear();
    }

    public void validateCart(int userId) throws UserNotLoggedException {
      User user = objectExchangeEM.find(User.class, userId);
      if(user == null) {
        throw new UserNotLoggedException();
      }

      for (Resource resource : resources) {
        if(resource.getUser().getId() != userId) {
          LentResource lentResource = new LentResource();
          lentResource.setResource(resource);
          lentResource.setUser(user);
          lentResource.setState(StateType.LOAN_WAITING);
          objectExchangeEM.persist(lentResource);

          // ... envoie d'email aux propriétaires ...
        }
      }
      resetCart();
    }

    public Collection<Resource> listCart() {
      return resources;
    }
}
```

Ce Session Bean représente le panier virtuel de l'application. Celui-ci étant spécifique à chaque client, nous devons utiliser un Session Bean de type `@Stateful`. En effet, l'état du panier doit être conservé d'une requête à une autre !

Les ressources choisies sont stockées dans la collection `resources`. Les méthodes `add`, `remove` et `listCart` permettent respectivement d'ajouter, supprimer une ressource et lister celles contenues dans le panier.

La méthode `validateCart()` valide le panier et crée les entrées de locations (`LentResource`) en leur affectant l'état « En cours de validation » (`LOAN_WAITING`).

11.3.4 Développement de la couche présentation

Une fois la couche métier réalisée, il reste alors à développer la couche présentation ainsi que la connexion entre celle-ci et la couche métier. Nous avons choisi d'utiliser JSF car cette technologie fait maintenant partie intégrante de la spécification Java EE.

Nous ne pouvons cependant pas nous attarder sur des explications pointues concernant le code de cette partie. Nous resterons précis et concis sur ces détails afin que vous puissiez les comprendre dans leur ensemble.

> **Remarque :** vous pouvez consulter tout le nécessaire sur JSF sur http://www.labosun.com/resource-fr-essentiels-1178-0-jsf.htm et sur http://java.sun.com/javaee/javaserverfaces/.

Création et configuration des Managed Beans

Le traitement des requêtes clientes (HTTP) est effectué, en JSF, grâce aux composants appelés « Managed Beans ». Ce sont de simples classes Java (POJO). Chacune de ces classes doit être déclarée dans le fichier de configuration JSF généralement nommé faces-config.xml afin d'être accessible (fig 11.19).

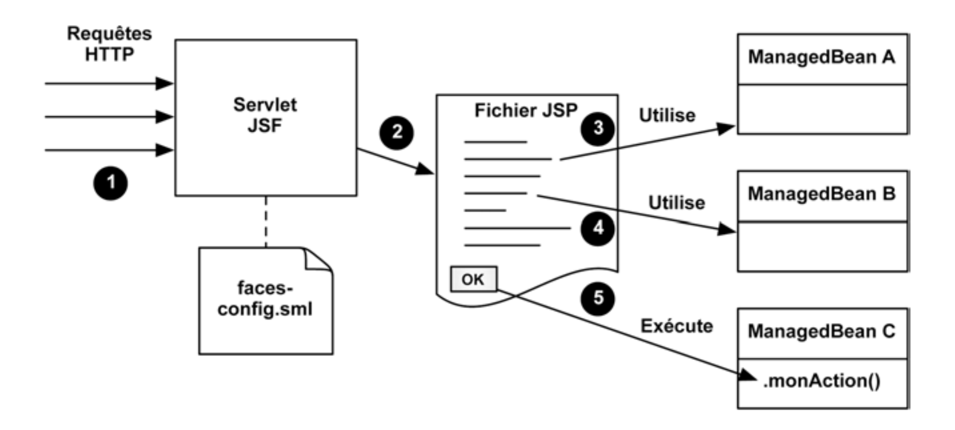


Figure 11.19 — Schéma simplifié du concept JSF

La figure 11.20 propose un schéma complet du processus de location d'objet présentant la liaison entre les éléments graphiques et les Managed Beans :

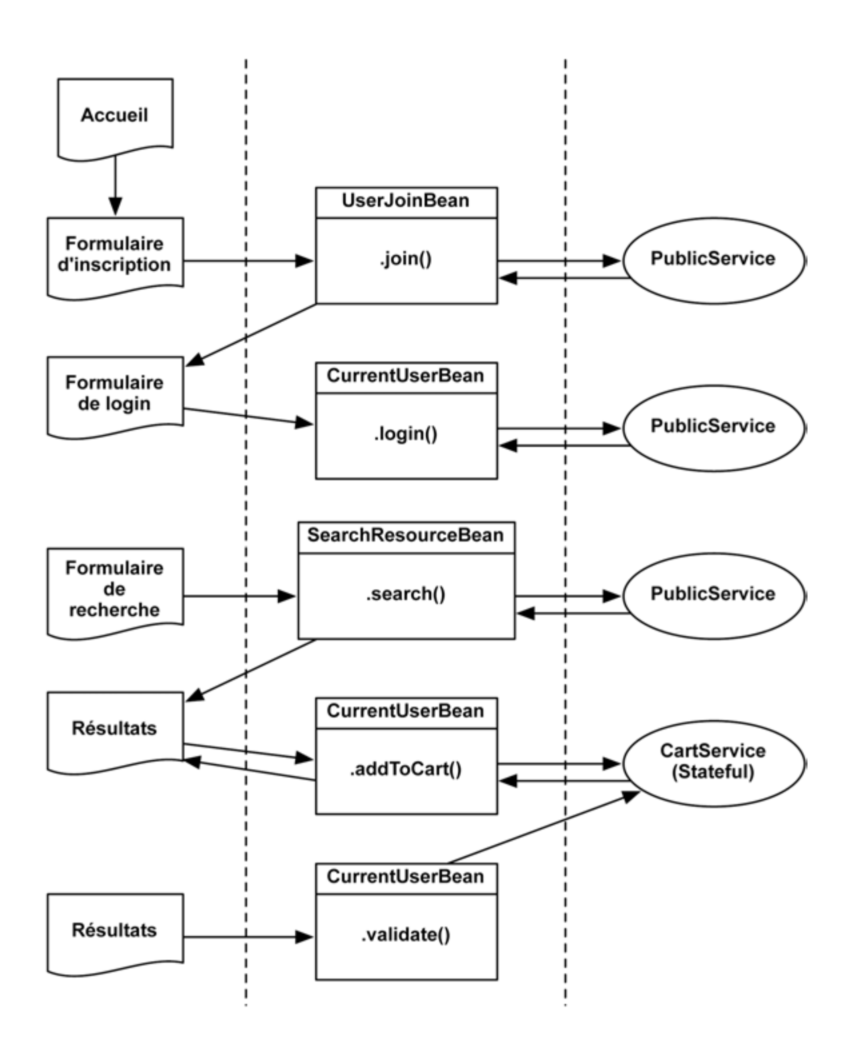


Figure 11.20 — Schéma processus de location

Le schéma précédent (fig. 11.20) utilise trois Managed Bean différents :

- `UserJoinBean` : gère les opérations liées à l'inscription des utilisateurs.
- `CurrentUserBean` : gère les opérations sur l'utilisateur courant (liée à la session courante).
- `SearchResourceBean` : gère les opérations de recherche et de récupération des ressources.

Voici le détail de `UserJoinBean` :

```
public class UserJoinBean {

  private User user;
```

```
  @EJB
  private PublicService publicService;

  public UserJoinBean() {
    setUser(new User());
  }

  public String join() {

    try {
      publicService.join(getUser());
    }
    catch(UserAlreadyJoinedException e) {
      // on reste sur la page du formulaire
      String msg = "L'utilisateur existe déjà";
      FacesMessage facesMsg = new FacesMessage(FacesMessage.SEVERITY_ERROR,
        msg, msg);
      FacesContext fc = FacesContext.getCurrentInstance();
      fc.addMessage("addError", facesMsg);
      return "";
    }
    catch(Exception e) {
      // on reste sur la page du formulaire
      String msg = "Nous avon rencontré des problèmes lors de l'ajout : " + e;
      FacesMessage facesMsg = new FacesMessage(FacesMessage.SEVERITY_ERROR,
        msg, msg);
      FacesContext fc = FacesContext.getCurrentInstance();
      fc.addMessage("addError", facesMsg);
      return "";
    }
    return "join_succeed";
  }

  public User getUser() {
    return user;
  }

  public void setUser(User user) {
    this.user = user;
  }
}
```

Nous pouvons remarquer dans le composant précédent, qu'il sert simplement de passerelle entre la requête et le Session Bean `PublicService`. En effet, ce dernier est injecté et appelé lorsque le formulaire d'inscription est validé.

Voici les détails du composant `SearchResourceBean` :

```
public class SearchResourceBean {

  @EJB()
  private PublicService publicService;

  private Resource resourceCriteria;

  private Category categoryCriteria;
```

```
private List<Resource> results;

private boolean searching;

public SearchResourceBean() {
  setResourceCriteria(new Resource());
}

protected int getCurrentResourceId() {
  ExternalContext ec = FacesContext.getCurrentInstance()
    .getExternalContext();
  HttpServletRequest request = (HttpServletRequest)ec.getRequest();
  String param = request.getParameter("id");
  return Integer.parseInt(param);
}

public String search() {
  setResults(publicService.searchResource(resourceCriteria));
  searching = true;
  return "public_search";
}

public Collection getListType() {
  Collection listSelect = new ArrayList();

  Iterator<Type> it = publicService.findAllType().iterator();
  while(it.hasNext()) {
    Type current = it.next();
    if(current.getId() > 0) {
      SelectItem selectItem = new SelectItem(current, current.getLabel());
      listSelect.add(selectItem);
    }
  }

  return listSelect;
}

public Collection getListCategory() {
  Collection listSelect = new ArrayList();

  Iterator<Category> it = publicService.findAllCategory().iterator();
  while(it.hasNext()) {
    Category current = it.next();
    if(current.getId() > 0) {
      SelectItem selectItem = new SelectItem(current, current.getLabel());
      listSelect.add(selectItem);
    }
  }

  return listSelect;
}

public Resource getResourceCriteria() {
  return resourceCriteria;
}

public void setResourceCriteria(Resource resourceCriteria) {
```

```
    this.resourceCriteria = resourceCriteria;
  }

  public Category getCategoryCriteria() {
    return categoryCriteria;
  }

  public void setCategoryCriteria(Category categoryCriteria) {
    this.categoryCriteria = categoryCriteria;
    resourceCriteria.getCategories().clear();
    if(categoryCriteria != null && categoryCriteria.getId() > 0) {
       resourceCriteria.getCategories().add(categoryCriteria);
    }
  }

  public List<Resource> getResults() {
    List<Resource> results = null;
    if(searching == false) {
      results = publicService.findLastResource();
    }
    else {
      results = publicService.searchResource(resourceCriteria);
    }

    // récupération de l'objet "currentUser"
    CurrentUserBean currentUser = (CurrentUserBean) FacesContext
     .getCurrentInstance().getApplication().getELResolver().getValue(
       FacesContext.getCurrentInstance().getELContext(), null, "currentUser");

    // supprime les objets déjà choisis
    results.removeAll(currentUser.getListCart());

    return results;
  }

  public void setResults(List<Resource> results) {
    this.results = results;
  }

  public Category getNullCategory() {
    return new Category();
  }

  public Type getNullType() {
    return new Type();
  }

  public List getCurrentResourceCategories() {
    return new LinkedList(currentResource.getCategories());
  }

}
```

Ce composant peut sembler plus complexe que le précédent à première vue. Il joue toutefois le même rôle de passerelle. Cette fois-ci il s'occupe de récupérer les critères de recherche à utiliser pour lister les différentes ressources disponibles. La

méthode `search()` est appelée lorsqu'un client valide le formulaire de recherche. Celle-ci renvoie vers la page d'affichage des résultats qui utilise la propriété `results` (*via* le *getter* `getResults()`) pour récupérer ces résultats.

Voici les détails du composant `CurrentUserBean` :

```
public class CurrentUserBean {

  private boolean logged = false;

  private User userLogged;

  @EJB()
  private PublicService publicService;

  @EJB()
  private CartService cart;

  public CurrentUserBean() {
    setUserLogged(new User());
  }

  public String login() {
    try {
      userLogged = publicService.login(userLogged.getEmail(),
        userLogged.getPassword());
      logged = true;
    }
    catch(UserNotLoggedException e) {
      // on reste sur la page du formulaire
      String msg = "Impossible de vous connecter, "
        +"veuillez vérifier vos identifiants";
      FacesMessage facesMsg = new FacesMessage(msg);
      FacesContext fc = FacesContext.getCurrentInstance();
      fc.addMessage("loginForm", facesMsg);
    }
    catch(Exception e) {
      // on reste sur la page du formulaire
      String msg = "Impossible de contacter le service d'authentification : "
        + e;
      FacesMessage facesMsg = new FacesMessage(msg);
      FacesContext fc = FacesContext.getCurrentInstance();
      fc.addMessage("loginForm", facesMsg);
    }
    return "";
  }

  public String deco() {
    setUserLogged(new User());
    logged = false;
    return "public_index";
  }

  public int getRequestId() {
    ExternalContext ec = FacesContext.getCurrentInstance()
      .getExternalContext();
```

```
    HttpServletRequest request = (HttpServletRequest)ec.getRequest();
    String param = request.getParameter("id");
    return Integer.parseInt(param);
  }

  public String add() {
    cart.add(getRequestId());
    return "";
  }

  public void removeFromCart() {
    cart.remove (getRequestId());
  }

  public String validCart() {
    try {
      cart.validateCart(userLogged.getId());
    }
    catch(UserNotLoggedException e) {
      String msg = "Vous devez vous connecter pour valider votre panier";
      FacesMessage facesMsg = new FacesMessage(FacesMessage.SEVERITY_ERROR,
        msg, msg);
      FacesContext fc = FacesContext.getCurrentInstance();
      fc.addMessage("validCartError", facesMsg);
      return "public_cart";
    }
    catch(Exception e) {
      String msg = "Une erreur est survenue lors de la validation du panier";
      FacesMessage facesMsg = new FacesMessage(FacesMessage.SEVERITY_ERROR,
        msg, msg);
      FacesContext fc = FacesContext.getCurrentInstance();
      fc.addMessage("validCartError", facesMsg);
      return "public_cart";
    }
    return "public_cart_validated";
  }

  public Collection getListCart() {
    return cart.listCart();
  }

  public boolean isLogged() {
    return logged;
  }

  public void setLogged(boolean logged) {
    this.logged = logged;
  }

  public User getUserLogged() {
    return userLogged;
  }

  public void setUserLogged(User userLogged) {
    this.userLogged = userLogged;
  }
}
```

Ce composant est le point central pour la sauvegarde des données entre le client et le serveur. Il regroupe sur l'utilisateur (connecté ou non), la référence vers le Stateful Session Bean et les méthodes de gestion du panier virtuel (`add()`, `removeFromCart()`, `validateCart()`).

Pour pouvoir utiliser ces composants dans les pages JSP, il faut les déclarer dans le fichier de config JSF faces-config.xml. Voici les lignes à ajouter à ce fichier précédemment créé.

```
<!-- managed beans session -->
<managed-bean>
  <managed-bean-name>currentUser</managed-bean-name>
  <managed-bean-class>
    com.labosun.objectexchange.application.CurrentUserBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>searchResource</managed-bean-name>
  <managed-bean-class>
    com.labosun.objectexchange.application.SearchResourceBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>


<!-- managed beans request -->
<managed-bean>
  <managed-bean-name>joinUser</managed-bean-name>
  <managed-bean-class>
    com.labosun.objectexchange.application.UserJoinBean
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

La balise `<managed-bean-scope>` définit le contexte auquel doit être associé le composant (`session`, `request`, `application`).

Les autres éléments à déclarer sont les règles de navigation. Vous avez sans doute remarqué que certaines méthodes retournent une chaîne de caractère (`String`). Cette chaîne retournée est évaluée par JSF et permet de choisir la vue à retourner au client. Dans le cas d'une méthode retournant une chaîne de caractère vide ou de type `void`, JSF réaffiche la page par laquelle l'utilisateur a fait sa requête. Voici le détail des règles de navigation utilisées dans la partie présentée.

```
<navigation-rule>
  <from-view-id>/*</from-view-id>
  <navigation-case>
    <from-outcome>public_index</from-outcome>
    <to-view-id>/public_index.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>public_search</from-outcome>
    <to-view-id>/public_listresource.jsp</to-view-id>
```

```
    </navigation-case>
    <navigation-case>
      <from-outcome>public_detail</from-outcome>
      <to-view-id>/public_detail.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>join</from-outcome>
      <to-view-id>/public_join.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>error</from-outcome>
      <to-view-id>/error.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>public_cart</from-outcome>
      <to-view-id>/public_cart.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>public_cart_validated</from-outcome>
      <to-view-id>/public_cart_validated.jsp</to-view-id>
    </navigation-case>

  </navigation-rule>

  <!-- rules -->

  <!-- inscription -->
  <navigation-rule>
    <from-view-id>/public_join.jsp</from-view-id>
    <navigation-case>
      <from-outcome>join_succeed</from-outcome>
      <to-view-id>/public_join_succeed.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
```

Chaque règle est définie par la balise `<navigation-rule>`. La balise `<from-view-id>` déclare le point d'entrée auquel la règle correspond (pattern d'url correspondant à la requête). Les balises `<navigation-case>` définissent les vues (`case`) disponibles pour la règle (en *mappant* un nom et la page associée). La balise `<from-outcome>` définit la chaîne de caractère à utiliser comme valeur de retour dans le Managed Bean exécuté.

Création des pages JSP

Les pages JSP correspondent à la génération de la vue (HTML). Il existe une correspondance avec les Managed Beans présentés précédemment. Voici l'ensemble des JSP utilisées :

- public_index.jsp : page d'accueil.
- public_join.jsp : page d'inscription
- public_join_succeed.jsp : page de confirmation d'inscription réussie
- public_listresource.jsp : page des résultats de ressources de la recherche
- public_detail.jsp : page de détail d'une ressource

- public_cart.jsp : page de récapitulatif du panier
- public_cart_validated.jsp : page de confirmation de validation du panier.

Voici le détail du contenu de la page public_join.jsp :

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:s="http://jboss.com/products/seam/taglib"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf"
  xmlns:h="http://java.sun.com/jsf/html">
  <body>
    <ui:composition template="/WEB-INF/template/template_public.jsp">
      <ui:define name="body">

        <h2><h:outputText value="Inscription" /></h2>

        <p><em>
          <h:outputText value="Etat civil et informations personnelles" />
        </em></p>


        <p>
          L'inscription vous permet de vous connecter sur le site,
          partager et louer des ressources.
        </p>

        <div class="cboxInfo">
          <h:form id="joinForm" >
            <h:messages style="color: green" layout="table"/>
            <h:panelGrid rowClasses="cboxForm" columns="2">

              <h:outputText value="Prenom :"/>
              <h:panelGroup>
                <h:inputText id="firstName"
                  value="#{joinUser.user.firstName}"
                  title="Firstname" required="true"/>
                <h:message for="firstName" styleClass="cboxError" />
              </h:panelGroup>

              <h:outputText value="Nom :"/>
              <h:panelGroup>
                <h:inputText id="lastName"
                  value="#{joinUser.user.lastName}"
                  title="Lastname" required="true"/>
                <h:message for="lastName" styleClass="cboxError" />
              </h:panelGroup>


              <h:outputText value="Email (login) :"/>
              <h:panelGroup>
                <h:inputText id="email"
                  value="#{joinUser.user.email}"
                  title="Email" required="true"/>
                <h:message for="email" styleClass="cboxError" />
              </h:panelGroup>
```

```
            <h:outputText value="Mot de passe :"/>
            <h:outputText value="Le mot de passe est auto-genere"/>

            <h:outputText value="Adresse :"/>
            <h:panelGroup>
              <h:inputText id="address"
                value="#{joinUser.user.address}"
                title="Address" required="true" />
              <h:message for="address" styleClass="cboxError" />
            </h:panelGroup>

            <h:outputText value="Code postal  :"/>
            <h:panelGroup>
              <h:inputText id="zipCode"
                value="#{joinUser.user.zipCode}"
                title="ZipCode" required="true" />
              <h:message for="zipCode" styleClass="cboxError" />
            </h:panelGroup>

            <h:outputText value="Ville :"/>
            <h:panelGroup>
              <h:inputText id="city"
                value="#{joinUser.user.city}"
                title="City" required="true" />
              <h:message for="city" styleClass="cboxError" />
            </h:panelGroup>

            <h:outputText value="Pays : "/>
            <h:panelGroup>
              <h:inputText id="country"
                value="#{joinUser.user.country}"
                title="Country" required="true" />
              <h:message for="country" styleClass="cboxError" />
            </h:panelGroup>

            <h:commandButton action="#{joinUser.join}" value="S'inscrire"/>

          </h:panelGrid>
        </h:form>
      </div>
    </ui:define>
  </ui:composition>
 </body>
</html>
```

Ce code définit un formulaire `joinForm` contenant l'ensemble des champs représentant un compte utilisateur. Voici le détail des principales balises JSF utilisées :

- `<h:form>` : déclare un formulaire. C'est un élément essentiel pour pouvoir envoyer des informations rentrées par l'utilisateur au serveur.
- `<h:outputText>` : permet d'afficher un texte et de le formater si besoin.
- `<h:inputText>` : représente un champ texte (`<input type="texte">`)
- `<h:message>` : permet d'afficher un message généré par le serveur. Nous utilisons cette balise généralement pour afficher les messages d'erreur.

- `<h:commandButton>` : génère un bouton servant à envoyer le formulaire et appeler la méthode passée par l'attribut `action`.

Figure 11.21 — Page d'inscription

Voici le détail du contenu de la page public_join_succeed.jsp (fig. 11.21) :

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:s="http://jboss.com/products/seam/taglib"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf"
  xmlns:h="http://java.sun.com/jsf/html">
  <body>
```

```
    <ui:composition template="/WEB-INF/template/template_public.jsp">
      <ui:define name="body">

        <h2><h:outputText value="Inscription ok !" /></h2>

        <p><em>
          <h:outputText value="Vous pouvez maintenant vous connecter" />
        </em></p>


        <p>
          L'inscription est réussie vous pouvez désormais vous connecter.
        </p>

      </ui:define>
    </ui:composition>
  </body>
</html>
```

Cette page affiche uniquement un message de confirmation d'inscription.

Voici le détail du contenu de la page public_listresource.jsp :

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:c="http://java.sun.com/jstl/core">

  <body>
    <ui:composition template="/WEB-INF/template/template_public.jsp">

      <ui:param name="page" value="public_object" />

      <ui:define name="body">
        <h:form >
          <h2>Objets disponibles : </h2>
          <p>
            <ui:include src="/WEB-INF/incl/tab_resourceresults.jsp" >
              <ui:param name="results" value="#{searchResource.results}"/>
            </ui:include>
            <br/>
          </p>
        </h:form>
      </ui:define>

    </ui:composition>
  </body>
</html>
```

Cette page affiche la liste des ressources retournées par la méthode `getResults()` du Managed Bean `searchResource`. La génération du code HTML de la liste des ressources est réalisée par la page `tab_resourceresults.jsp`. Celle-ci est appelée par la balise `<ui:include>` et la collection de ressources est passée *via* le paramètre `param` (balise `<ui:param>`).

ObjectExchange

Accueil S'incrire Objets Panier

Recherche

Titre :

Auteur :

Details :

Type : Tout type

Categories : Toute categorie

Rechercher

Login

Login :

Pass

Se connecter

Creer un compte

Votre panier

Recapitulatif

Valider

Liste des objets disponibles

Voici la liste des ressources disponibles pour la location.

Tableau des ressources :

Action	Image	Type	Titre	Auteur	Date
Ajoute Detail		LIVRE	EJB3 la révolution	Laboratoire Sun de SUPINFO	09/04/2006
Ajoute Detail		DVD	Loving Java ..	PopomCorp	05/05/2006
Ajoute Detail		CD	Musique de Film Volume 1	Universal Music	01/06/2006
Ajoute Detail		LIVRE	JBoss Seam par la pratique	Cyril Pollet	21/06/2006

Figure 11.22 — Page liste des ressources

Voici le détail du contenu de la page tab_resourceresults.jsp (fig. 11.22) :

```
<p xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:c="http://java.sun.com/jstl/core">

  <h:dataTable value="#{results}" styleClass="normaltable"
    var="item" border="0" cellpadding="1" cellspacing="0">
    <h:column>
      <f:facet name="header">
        <h:outputText value="Action"/>
```



```
      </f:facet>

      <h:commandLink action="#{currentUser.add}">
        <f:param name="id" value="#{item.id}"/>
        <h:outputText value="Ajoute"/>
      </h:commandLink>
      <br/>
      <h:commandLink action="#{searchResource.detail}">
        <f:param name="id" value="#{item.id}"/>
        <h:outputText value="Detail"/>
      </h:commandLink>
    </h:column>

    <h:column>
      <f:facet name="header">
        <h:outputText value="Type"/>
      </f:facet>
      <h:outputText value="#{item.type.label}"/>
    </h:column>


    <h:column>
      <f:facet name="header">
        <h:outputText value="Titre"/>
      </f:facet>
      <h:outputText value="#{item.title}"/>
    </h:column>

    <h:column>
      <f:facet name="header">
        <h:outputText value="Auteur"/>
      </f:facet>
      <h:outputText value="#{item.author}"/>
    </h:column>

    <h:column>
      <f:facet name="header">
        <h:outputText value="Date"/>
      </f:facet>
      <h:outputText value="#{item.date}">
        <f:convertDateTime pattern="dd/MM/yyyy"/>
      </h:outputText>
    </h:column>

  </h:dataTable>
</p>
```

Nous utilisons la balise `<h:datatable>` pour définir le tableau que l'on souhaite générer. Celui-ci utilise la valeur contenue dans `#{results}`, c'est-à-dire la collection de ressources. La balise `<h:column>` définit une nouvelle colonne au tableau, nous affichons cinq colonnes (Action, Type, Titre, Auteur, Date). La définition des titres de colonnes s'effectue *via* la balise `<f:facet>` et l'attribut `name`.

Figure 11.23 — Page de détail d'une ressource

Voici le détail du contenu de la page public_detail.jsp (fig. 11.23):

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:c="http://java.sun.com/jstl/core">

  <body>
    <ui:composition template="/WEB-INF/template/template_public.jsp">

      <ui:param name="page" value="public_object" />
```



```
      <ui:define name="body">

        <h:form >
          <h2>Details de l'objet :
            #{searchResource.currentResource.title}</h2>
          <p>
            Details : <br/>
            #{searchResource.currentResource.details}<br/>
            <br/>
            Type :<br/>
            <h:outputText
              value="#{searchResource.currentResource.type.label}"/>
            <br/>
            Categories : <br/>
            <c:forEach var="cat"
              items="#{searchResource.currentResourceCategories}">
              #{cat.label}<br/>
            </c:forEach>
            <br/>
            <h:commandLink action="#{currentUser.add}">
              <f:param name="id"
                value="#{searchResource.currentResource.id}"/>
              <h:outputText value="Ajouter au panier"/>
            </h:commandLink>
            <br/>
          </p>
        </h:form>

      </ui:define>
    </ui:composition>
  </body>
</html>
```

Cette page détaille une ressource lorsque l'utilisateur clique sur le lien « Détails » du tableau précédent. Nous utilisons ici la balise `<c:forEach>` pour afficher l'ensemble des catégories associées à cette ressource.

Figure 11.24 — Page récapitulatif du panier

Voici le détail du contenu de la page public_cart.jsp (fig. 11.24) :

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:c="http://java.sun.com/jstl/core">

  <body>
  <ui:composition template="/WEB-INF/template/template_public.jsp">

      <ui:param name="page" value="public_cart" />

      <ui:define name="body">
```



```
<h1>Panier virtuel</h1>

<p>
  Voici le contenu de votre panier de locations.
</p>

<h2>Recapitulatif des locations souhaitees :</h2>
<p>
  <h:messages style="color: green" layout="table"/>
  <h:form id="cartForm" >
    <h:dataTable value="#{currentUser.listCart}" styleClass="dvdtable"
      var="item" border="0" cellpadding="1" cellspacing="0">

      <h:column>
        <f:facet name="header">
          <h:outputText value="Type"/>
        </f:facet>
        <h:outputText value="#{item.type.label}"/>
      </h:column>


      <h:column>
        <f:facet name="header">
          <h:outputText value="Titre"/>
        </f:facet>
        <h:outputText value="#{item.title}"/>
      </h:column>

      <h:column>
        <f:facet name="header">
          <h:outputText value="Details"/>
        </f:facet>
        <h:outputText value="#{item.details}"/>
      </h:column>

      <h:column>
        <f:facet name="header">
          <h:outputText value="Auteur"/>
        </f:facet>
        <h:outputText value="#{item.author}"/>
      </h:column>

      <h:column>
        <f:facet name="header">
          <h:outputText value="Date"/>
        </f:facet>
        <h:outputText value="#{item.date}">
          <f:convertDateTime pattern="dd/MM/yyyy"/>
        </h:outputText>
      </h:column>


      <h:column>
        <h:commandLink action="#{currentUser.removeFromCart}">
          <f:param name="id" value="#{item.id}"/>
          <h:outputText value="Annuler"/>
        </h:commandLink>
```

```
          </h:column>
        </h:dataTable>
        <br/>
        <br/>
        <h:commandLink action="#{currentUser.validCart}">
          <h:outputText value="Valider la selection"/>
        </h:commandLink>
      </h:form>
    </p>
  </ui:define>
 </ui:composition>
</body>
</html>
```

Cette page récapitule les ressources sélectionnées pour la location. Nous utilisons une fois de plus la balise `<h:datatable>` pour générer le tableau. Vous pouvez remarquer l'utilisation de la balise `<f:convertDateTime>` permettant de formater l'affichage de la date d'ajout de la ressource.

Voici le détail du contenu de la page public_cart_validated.jsp :

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:c="http://java.sun.com/jstl/core">

  <body>
    <ui:composition template="/WEB-INF/template/template_public.jsp">

      <ui:param name="page" value="public_cart" />

      <ui:define name="body">

        <h1>Panier virtuel valide</h1>

        <p>
          Veuillez attendre la validation de vos demandes par les
          Propriétaires respectifs.
        </p>
      </ui:define>
    </ui:composition>
  </body>
</html>
```

Nous avions parlé au début de cette partie de quelques fichiers se trouvant dans WEB-INF/incl. Ces fichiers sont en fait des modules graphiques partagés entre les différentes pages vues précédemment.

Voici le détail du contenu de la page login.jsp :

```
<c:choose xmlns="http://www.w3.org/1999/xhtml"
  xmlns:s="http://jboss.com/products/seam/taglib"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
```

```
xmlns:c="http://java.sun.com/jstl/core">

<c:when test="#{currentUser.logged==false}">
  <div class="box">
    <h:form id="loginForm" >
      <dl>
        <dt class="boxHeader">Login</dt>
        <dd class="boxContent">
          <h:message for="loginForm" layout="table" />
          <dl>
            <dt>Login : </dt>
            <dd>
              <h:inputText value="#{currentUser.userLogged.email}"
              size="16" />
            </dd>
            <dt>Pass </dt>
            <dd>
              <h:inputSecret value="#{currentUser.userLogged.password}"
              size="16" />
            </dd>
            <dd>
              <h:commandButton action="#{currentUser.login}"
                value="Se connecter" class="formButton"
                style="width: 166px;"/>
            </dd>

          </dl>
        </dd>
      </dl>
      <dl>
        <dt class="boxFooter">
          <h:commandLink action="#{currentUser.join}">
            <h:outputText  value="Creer un compte"/>
          </h:commandLink>
        </dt>
      </dl>
    </h:form>
  </div>
</c:when>

<c:otherwise>
  <div class="box">
    <dl>
      <dt class="boxHeader">Bienvenu,
        #{currentUser.userLogged.firstName}</dt>
      <dd class="boxContent">
        <h:form>
          <dl>
            <dd>Vous êtes connecté !</dd>

            <dd>
              <h:commandButton action="#{currentUser.deco}"
              value="Deconnexion" class="formButton" style="width: 166px;"/>
            </dd>
          </dl>
        </h:form>
      </dd>
```

```
      </dl>
    </div>
  </c:otherwise>

</c:choose>
```

Cette page affiche le formulaire de login si la personne n'est pas connectée ou des informations la concernant dans l'autre cas. Pour cela, nous utilisons la balise JSTL `<c:choose>` qui correspond en quelque sorte à un `switch` en Java.

Voici le détail du contenu de la page little_cart.jsp :

```
<div xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:c="http://java.sun.com/jstl/core" class="box" >
  <!-- panier (petit) -->

  <h:form id="littleCartForm" >
    <dl>
      <dt class="boxHeader">Votre panier</dt>
      <dd class="boxContent">
        <dl>
          <dt><h:outputText value="Recapitulatif"/></dt>
          <dd>
            <h:dataTable value="#{currentUser.listCart}"
              var="item" border="0" cellpadding="1" cellspacing="0">
              <h:column>
                <h:outputText value="#{item.title}"/>
              </h:column>

              <h:column>
                <h:commandLink action="#{currentUser.removeFromCart}">
                  <f:param name="id" value="#{item.id}"/>
                  <h:outputText value="Annuler"/>
                </h:commandLink>
              </h:column>
            </h:dataTable>
          </dd>
          <dd>
            <h:commandButton action="#{currentUser.validCart}"
              value="Valider" class="formButton" style="width: 166px;"/>
          </dd>
        </dl>
      </dd>
    </dl>
  </h:form>
</div>
```

Voici le détail du contenu de la page search_resource.jsp :

```
  <div xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:c="http://java.sun.com/jstl/core" class="box" >
```

```
<!-- recherche -->

<h:form id="searchResForm" >
  <dl>
    <dt class="boxHeader">Recherche</dt>
    <dd class="boxContent">
      <h:message for="loginForm" layout="table" />
      <dl>
        <dt><h:outputText value="Titre :"/></dt>
        <dd><h:inputText id="title"
          value="#{searchResource.resourceCriteria.title}"
          title="Title" size="20"/>
        </dd>

        <dt><h:outputText value="Auteur :"/></dt>
        <dd><h:inputText id="author"
          value="#{searchResource.resourceCriteria.author}"
          title="Author" size="20"/>
        </dd>

        <dt><h:outputText value="Details :"/></dt>
        <dd><h:inputText id="details"
          value="#{searchResource.resourceCriteria.details}"
          title="Details" size="20" />
        </dd>
        <dt><h:outputText value="Type :"/></dt>
        <dd>

          <h:selectOneMenu id="type"
            value="#{searchResource.resourceCriteria.type}"
            title="Type" converter="typeConverter" >

            <f:selectItem  itemLabel="Tout type"
              itemValue="#{searchResource.nullType}" />
            <f:selectItems  id="selecttype"
              value="#{searchResource.listType}"  />

          </h:selectOneMenu>
        </dd>

        <dt><h:outputText value="Categories :"/></dt>
        <dd><h:selectOneMenu id="categories"
          value="#{searchResource.categoryCriteria}"
          title="Categories" converter="categoryConverter" >
          <f:selectItem  itemLabel="Toute categorie"
            itemValue="#{searchResource.nullCategory}"/>
          <f:selectItems  id="selectcategory"
            value="#{searchResource.listCategory}"/>
        </h:selectOneMenu>
        </dd>

        <dd><h:commandButton action="#{searchResource.search}"
          value="Rechercher" class="formButton" style="width: 166px;"/>
        s</dd>
      </dl>
    </dd>
  </dl>
```

```
  </h:form>
</div>
```

Certes simple, cette application n'en est pas moins complète. Nous avons pu, grâce à celle-ci, illustrer de manière appliquée les principaux concepts liés à une architecture Java EE et EJB 3. Vous avez désormais toutes les clés en main pour mener à bien le développement d'une telle application. À vous de jouer !

En résumé

De l'analyse au développement, l'application ObjectExchange permet d'illustrer les concepts étudiés tout au long de cet ouvrage et vous permet de mieux appréhender le développement d'un processus complet. Ce chapitre a également permis de présenter comment connecter les couches métiers (EJB) et présentations (JSF). Nous pouvons tout de même remarquer qu'il existe une certaine redondance entre les Managed Beans et les Session Beans : c'est un des inconvénients de JSF. Toutefois, des solutions plus efficaces arrivent sur le marché. La plus mature, à ce jour, est sans aucun doute le *framework* JBoss Seam (voir paragraphe 10.4) qui ouvre de très bonnes perspectives d'avenir aux développeurs Java EE.

L’avenir d’EJB 3

Nous espérons que cet ouvrage vous aura permis de comprendre l’enjeu des EJB 3, qui est actuellement la technologie la plus prometteuse pour la couche métier des applications d’entreprise.

Comme vous avez pu le constater, cette technologie a grandement simplifié le développement de la couche métier, en grande partie grâce aux évolutions du JDK 1.5. Cette simplicité apparente n’a pas pour autant réduit la puissance des EJB. Bien au contraire, les serveurs d’applications compatibles EJB 3 tiennent bien mieux la charge et permettent une optimisation extrêmement poussée.

Il faut néanmoins garder à l’esprit que c’est une technologie dont les spécifications ont été validées il y a peu de temps, et qu’elle doit encore faire ses preuves en entreprise. Certes attirante par sa simplicité de mise en œuvre, un grand nombre d’entreprises sont encore « frileuses » face à une éventuelle migration de leur infrastructure applicative, et la jeunesse de cette technologie ne joue pas en sa faveur.

EJB 3 n’en est pas moins plébiscité par la communauté. Il existe déjà plusieurs plates-formes stables utilisant cette technologie. La plus avancée, lors de l’écriture de cet ouvrage, est JBoss Seam, présenté dans le chapitre 10. D’autres plates-formes de ce type sont bien évidemment à prévoir dans les prochains mois.

Java Entreprise Edition 5.0, et plus particulièrement EJB 3, est une évolution importante de la plate-forme Java, mais est-ce réellement une révolution, comme le soutient une grande partie de la communauté ? L’avenir nous le dira.

Rappel : N’hésitez pas à consulter notre site *www.labo-sun.com* pour y suivre l’actualité du monde Java et parcourir les divers articles technologiques.

Index

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

050623 – (I) – (1,5) – OSB 100° – STY – CDD

Achevé d'imprimer sur les presses de
SNEL Grafics sa
Z.I. des Hauts-Sarts - Zone 3
Rue Fond des Fourches 21 – B-4041 Vottem (Herstal)
Tél +32(0)4 344 65 60 - Fax +32(0)4 289 99 61
octobre 2006 – 40242

Dépôt légal : novembre 2006
Imprimé en Belgique

InfoPro

TYPE D'OUVRAGE

L'ESSENTIEL	SE FORMER	RETOURS D'EXPÉRIENCE



MANAGEMENT DES SYSTÈMES D'INFORMATION

APPLICATIONS MÉTIERS

ÉTUDES, DÉVELOPPEMENT, INTÉGRATION

EXPLOITATION ET ADMINISTRATION

RÉSEAUX & TÉLÉCOMS

Laboratoire SUPINFO
des technologies Sun

EJB 3

Des concepts à l'écriture du code
Guide du développeur

Cet ouvrage est fondé sur les cours dispensés par le laboratoire SUPINFO des technologies Sun. Son objectif est de présenter et d'illustrer la nouveauté majeure de la dernière version de la plate-forme Java Entreprise : **EJB 3**.

Il a été conçu comme un guide de formation et un support de référence répondant aux questions concrètes de développement et de maintenance des services et des objets métiers. Il présente en complément une étude pratique complète basée sur un cas réel de développement d'une application entreprise.

Cet ouvrage conviendra aux développeurs Java désireux de s'initier aux systèmes de persistance des données, aux développeurs EJB 2 souhaitant évoluer vers un système plus modulable et plus rapide à mettre en place ainsi qu'aux développeurs J2EE cherchant à la fois un système performant de persistance de données et un guide pratique proposant de **nombreux exemples de code prêt à l'emploi**.

SUPINFO

est un établissement d'enseignement supérieur qui forme des ingénieurs informaticiens de haut niveau. L'école a établi des partenariats pédagogiques très forts avec les plus grands noms du secteur informatique mondial (Microsoft, Oracle, IBM, Cisco Systems, Sun Microsystems, Mandriva, Apple Computer…) et a créé des laboratoires pédagogiques francophones.

Au travers de son portail *www.labo-sun.com* et de différentes contributions, le **laboratoire SUPINFO des technologies Sun** est aujourd'hui une référence pédagogique reconnue.



Le code source des exemples et des applications est téléchargeable sur le site *www.labo-sun.com*.







ISBN 2 10 050623 4

www.dunod.com